H.E.M.E.S. – Informatique

André CLARINVAL

Le langage C

édition : octobre 1998

Chapitre 1. Survol introductif

1. Définitions de départ

Algorithme : *méthode* de *composition d'opérations*
pour arriver à la solution *certaine*
de tout problème appartenant à une *classe* bien *définie*.

Programme : exécution d'algorithme(s) par l'ordinateur.[1]
(cette définition est partielle)

La démarche créative des informaticiens comporte principalement deux étapes :

- l'**analyse** : définition des classes de problèmes à résoudre et des méthodes de résolution;
- la **programmation** : mise des algorithmes en forme de programmes.

2. Etape 1 : élaboration de l'algorithme (analyse)

2.1. Etape 1.1 : définir la classe de problèmes

Soit le **problème** "élever le nombre 6 à la puissance 5"; il y aurait peu d'intérêt à créer un programme pour résoudre ce problème non répétitif : le temps de création du programme serait bien supérieur au temps de résolution du problème. Généralisons donc, et envisageons la **classe de problèmes** "élever un nombre N à la puissance P".

Le fait de généraliser implique l'utilisation de **variables**. En *mathématiques*, une variable est un nom symbolique (N ou P) substitué à une valeur quelconque d'un certain domaine ou ensemble de valeurs. En *informatique*, ce nom **identificateur** est donné à une "case de mémoire" capable de contenir successivement (l'inscription de) différentes valeurs. Il est très important de distinguer la **valeur** (contenu de la case) et l'**adresse** (numéro de case) d'une variable.

ident.	a	n	x
adr.	15	16	17
valeur	234	-69	4.5

Une **constante** est une case de mémoire au contenu non modifiable pendant l'exécution du programme.

Dans les langages de programmation, une *variable* est toujours désignée par son *identificateur*, tandis que, le plus souvent, on indique directement la *valeur* d'une *constante*. On écrira par exemple : *i + 1*.

[1] Il existe également des programmes utilisant des méthodes heuristiques plutôt qu'algorithmiques. **Heuristique** : méthode de recherche d'une *solution incertaine*. Exemples d'applications : reconnaissance des formes (reconnaissance de la voix, de l'écriture manuscrite ...).

Opération fondamentale des ordinateurs, l'**affectation** consiste à inscrire une valeur dans une variable. Nous énoncerons une opération d'affectation sous la forme ***variable ← valeur***. Par exemple, *i ← i+1* exprime le fait d'ajouter 1 à la valeur actuelle de la variable `i` et de réinscrire dans `i` la nouvelle valeur. Cet énoncé comporte :

• une **expression de droite** (constante, variable, expression arithmétique ...) fournissant une *valeur*,
• une **expression de gauche** donnant l'*adresse* d'une variable (cas le plus simple : un nom de variable),
• un **opérateur** d'affectation (ici, le symbole ←) signifiant "inscrire dans la variable désignée par l'expression de gauche la valeur fournie par l'expression de droite".

La **classe** de problèmes est-elle **bien** (ou suffisamment) **définie** par l'énoncé "élever un nombre N à la puissance P" ? Non. Nous devons préciser la nature des arguments N et P. Posons que P doit être un nombre entier tandis que N peut être un nombre réel. P est-il signé ? Si oui, la méthode de calcul diffère selon que sa valeur est positive ou négative (multiplications ou divisions). Quant au résultat, puisque N est un nombre réel, il s'agira d'un nombre réel.

On dit que *entier, réel,* etc. sont des **types de données**, définissant à la fois le *domaine des valeurs* permises, le mode de *représentation en mémoire* et la gamme des *opérations* possibles sur ces données.

La définition d'une classe de problèmes doit toujours indiquer les **pré-conditions** qui rendent la solution possible. La délimitation précise du domaine de valeurs des variables utilisées fait partie de ces pré-conditions. Par exemple – question à se poser souvent – : la méthode de résolution vaudra-t-elle également si tel nombre vaut 0 ou ne faut-il pas plutôt exclure la valeur 0 ?

En résumé :

ALGORITHME :	PUISSANCE	
ARGUMENTS :	REEL :	N
	ENTIER :	P
RESULTAT :	REEL:	R

2.2. Etape 1.2 : définir la méthode de résolution

Existe-t-il une méthode ou une opération toute faite qui apporte la solution aux problèmes appartenant à la classe que nous venons de définir ? Supposons que non et que nous ne disposions que des **opérations de base** `+ - x /` . Il est donc nécessaire de combiner ces opérations dans un **algorithme**.

Ceci suppose l'existence de **mécanismes de composition** des opérations. Voici les plus courants.

• La **séquence** est une suite d'opérations effectuées l'une après l'autre, en suivant l'ordre de leur mention.

DEBUT *opération 1*
opération 2
.....
FIN

• L'**itération** consiste à répéter l'exécution d'une séquence d'opérations, tant qu'une condition reste vraie; cette condition est testée avant chaque exécution de la séquence. Pour que la répétition finisse par s'arrêter, la séquence répétée doit contenir des opérations capables de rendre la condition fausse. Si, dès le départ, la condition est fausse, la séquence d'opérations n'est pas exécutée.

TANT QUE *condition*
EXECUTER *séquence*
FIN-TANT

• La **sélection** choisit parmi plusieurs une seule séquence d'opérations, et l'exécute; le choix est dicté par une condition. Une des séquences possibles peut être vide de toute opération; si cette séquence est sélectionnée, aucune opération n'est donc exécutée.

SI *condition*
ALORS *séquence*
SINON *séquence*
FIN-SI

Première version de l'algorithme :

```
ALGORITHME : PUISSANCE

ARGUMENTS :  REEL ≠ 0 :      N
             ENTIER :        P
RESULTAT :   REEL :          R

METHODE :    SI P = 0
             ALORS R ← 1.0
             FIN-SI
             SI P > 0
             ALORS DEBUT R ← 1.0
                         exécuter P fois "R ← R x N"
                   FIN
             FIN-SI
             SI P < 0
             ALORS DEBUT R ← 1.0
                         exécuter –P fois "R ← R / N"
                   FIN
             FIN-SI
```

On donne le nom d'**instruction** à un énoncé qui, comme "R ← R x N", prescrit l'exécution de certaines opérations (*x* et ←).

Remarque. L'utilisation de la division R / N nous oblige à ajouter à la définition de la classe de problèmes la pré-condition N ≠ 0.

Deuxième version de l'algorithme.

Comment "exécuter moins P fois" quand P est négatif ? Créer un compteur ayant au départ la valeur de P et compter les exécutions en incrémentant ce compteur jusqu'à ce qu'il atteigne la valeur 0. Un mécanisme analogue peut être utilisé dans le cas de P positif.

```
ALGORITHME : PUISSANCE

ARGUMENTS : REEL ≠ 0 :      N
            ENTIER :        P
RESULTAT :  REEL :          R

METHODE :   SI P = 0
            ALORS R ← 1.0
            FIN-SI
            SI P > 0
            ALORS DEBUT R ← 1.0
                        I ← P    (compteur)
                        TANT QUE I > 0
                        EXECUTER     DEBUT R ← R x N
                                          I ← I – 1
                                     FIN
                        FIN-TANT
                  FIN
            FIN-SI
            SI P < 0
            ALORS DEBUT R ← 1.0
                        I ← P    (compteur)
                        TANT QUE I < 0
                        EXECUTER     DEBUT R ← R / N
                                          I ← I + 1
                                     FIN
                        FIN-TANT
                  FIN
            FIN-SI
```

On voit comment les structures de composition **s'emboîtent** les unes dans les autres : une construction alternative ou répétitive est elle-même considérée comme étant une opération.

Nous avons exprimé l'algorithme au moyen d'un certain formalisme. Les informaticiens disent qu'un texte comme celui-ci constitue un **pseudo-code**, rédigé dans un **pseudo-langage** ... langage *pseudo*- parce que ce formalisme n'a pas la rigueur des langages de programmation employés pour rédiger les programmes.

2.3. Etape 1.3 : optimiser l'algorithme

Ne peut-on améliorer cet algorithme ? le rendre plus lisible, supprimer des opérations parasites, rendre la méthode moins coûteuse en temps d'exécution, généraliser davantage la solution ... ?

Première optimisation. Dans les trois cas (d'après la valeur de P), l'algorithme ci-dessus commence par exécuter l'affectation "R ← 1". L'algorithme peut être modifié de manière telle que cette instruction soit rédigée une seule fois, au début.

```
METHODE :   DEBUT R ← 1.0
                SI P > 0
                ALORS DEBUT I ← P    (compteur)
                            TANT QUE I > 0
                            EXECUTER     DEBUT R ← R x N
                                                I ← I – 1
                                         FIN
                            FIN-TANT
                      FIN
                FIN-SI
                SI P < 0
                ALORS DEBUT I ← P    (compteur)
                            TANT QUE I < 0
                            EXECUTER     DEBUT R ← R / N
                                                I ← I + 1
                                         FIN
                            FIN-TANT
                      FIN
                FIN-SI
            FIN
```

Deuxième optimisation : supprimer les alternatives SI, qui font double emploi avec les tests effectués dans les constructions TANT QUE.

```
METHODE :   DEBUT R ← 1.0
                I ← P    (compteur)
                TANT QUE I > 0
                EXECUTER     DEBUT R ← R x N
                                   I ← I – 1
                             FIN
                FIN-TANT
                TANT QUE I < 0
                EXECUTER     DEBUT R ← R / N
                                   I ← I + 1
                             FIN
                FIN-TANT
            FIN
```

Troisième optimisation : utiliser directement P comme compteur d'itération. Ceci *implique qu'on n'ait plus besoin de la valeur de P après l'exécution de l'algorithme* !

```
METHODE :   DEBUT R ← 1.0
                TANT QUE P > 0
                EXECUTER     DEBUT R ← R x N
                                   P ← P – 1
                             FIN
                FIN-TANT
```

```
                TANT QUE P < 0
                EXECUTER    DEBUT R ← R / N
                                  P ←P + 1
                            FIN
                FIN-TANT
        FIN
```

3. Le concept de langage de programmation. Présentation du langage C

L'algorithme doit être rédigé dans un langage (un formalisme) compréhensible par l'ordinateur. Plus précisément, le langage en question doit être compréhensible à la fois par l'homme (le programmeur) et par un programme préexistant qui pourra analyser le texte et le convertir en "quelque chose" d'exécutable par l'ordinateur. Un tel programme s'appelle un **compilateur**.

Pour être analysable par l'automate qu'est le compilateur, un **langage** de programmation est extrêmement formalisé, et les règles d'écriture ou de **syntaxe** extrêmement précises.

Nous utiliserons dans ce cours le langage **C**, qui est un des plus couramment employés aujourd'hui.

Histoire du langage C

Le langage C a été défini en 1972-73 par l'Américain Ritchie, un des auteurs du système d'exploitation UNIX, pour servir à la rédaction des programmes composant ce système d'exploitation. Plusieurs particularités du langage sont révélatrices du genre de programmes que l'on visait : en proposant des équivalents pour la plupart des possibilités des langages assembleurs, le langage C ambitionnait de se substituer à ces derniers.

Le nom de "C" est l'aboutissement d'une suite de noms de langages qui s'inspirèrent successivement l'un l'autre : CPL ("Combined Programming Language"), BCPL ("Basic CPL"), B (avatar de BCPL), C (successeur de B). Plus tard, viendra C++ (C augmenté de nouvelles possibilités pour la programmation dite *par objets – "object oriented"*).

C est surtout un des nombreux langages qui se situent dans la lignée d'ALGOL ("ALGOrithmic Language"), ce langage qui, dès 1960, définissait et mettait en oeuvre les concepts fondamentaux de la programmation de style algorithmique.

Depuis les années 80, UNIX, qui n'était pas la propriété d'un fabricant d'ordinateurs particulier, est devenu le système d'exploitation d'un grand nombre d'ordinateurs, de toutes sortes de marques. Le succès d'UNIX a inévitablement entraîné le succès du langage C ...

La version originale du langage C a été décrite dans la première édition de l'ouvrage *The C Programming Language* publiée en 1978 par **Kernighan et Ritchie**. En 1989, l'**ANSI** (American National Standards Institute) définissait une version normalisée du langage C, qui contient quelques extensions par rapport à la version originale – la seconde édition de l'ouvrage de Kernighan et Ritchie décrit cette nouvelle version, sur laquelle nous nous appuierons.[2]

[2] Kernighan & Ritchie, *Le langage C ANSI*, 2ème édition, traduction française; éd. Masson, 1992.

4. Etape 2 : réalisation du programme

4.1. Etape 2.1 : transformer l'algorithme en fonction programmée

Le texte que nous allons rédiger constitue une fonction.

> Une **fonction** est une partie de programme qui reçoit des **arguments** ou **paramètres** (la racine N et l'exposant P) et, généralement, rend une valeur en **résultat**.

	les parties A.B. du texte d'une fonction doivent se suivre dans l'ordre illustré ici
`float puiss(float n,int p)`	A. déclaration (ou signature) de la **fonction** : *type_du_résultat nom (paramètres_formels)* les paramètres reçus par une fonction sont dits "formels" déclaration d'un **paramètre formel** : *type nom*
`/* élever n à la puissance p */`	entre /* */ exemple de **commentaire** un commentaire n'est pas analysé par le compilateur *des commentaires peuvent être placés n'importe où*
`{`	B. définition (ou corps) de la fonction sous la forme d'un **bloc** entre { } *les parties X.Y. du texte d'un bloc doivent se suivre dans l'ordre illustré ici*
`float resultat;`	X. déclaration des **variables locales**, c'est-à-dire accessibles aux seules opérations à l'intérieur du bloc
`resultat=1.0;` `while(p>0)` `{` `resultat=resultat*n;` `p=p-1;` `}` `while(p<0)` `{` `resultat=resultat/n;` `p=p+1;` `}` `return (resultat);` `}`	Y. opérations ou **instructions** certaines instructions peuvent contenir des blocs *habituellement,* ces blocs ne contiennent pas de déclarations de variables

Dans ce texte,

- les mots anglais sont des **mots clés** du langage C, définis par les créateurs de ce langage;
- les lettres isolées et les noms français sont les **identificateurs** des objets définis par le programmeur.

Quelques éléments du "vocabulaire" C (mots clés et opérateurs) :

- types de données : `int`*`eger`* nombre entier
  `float`*`ing point`* nombre réel en virgule flottante[3]
  (de valeur à valeur, la position de la virgule varie)

[3] Dans la notation américaine, les parties entière et décimale d'un nombre réel sont séparées par un point, plutôt que par une virgule; en outre, si la partie entière est nulle, elle peut être omise : `.50 = 0.50`.

– opérateurs :	d'affectation	=
	arithmétiques	+ – * (multiplication) /
	de comparaison	< > <= (inférieur ou égal) >= (supérieur ou égal)
		== (égal) != (différent)
– instructions :	`while`	TANT QUE
	`if`	SI
	`else`	SINON
	`return`	terminer l'exécution de la fonction
		et, s'il y a lieu, "retourner" le résultat

Règles et habitudes syntaxiques générales

La disposition du texte est libre, en ce sens qu'il n'existe pas de colonnes ou de marges.
Pour accroître la lisibilité, les programmeurs ont l'habitude de décaler vers la droite le contenu d'un bloc.

Les espaces sont facultatifs, sauf pour séparer deux mots.

Le caractère `;` est un terminateur de déclaration ou d'instruction.
On écrit habituellement sur une ligne une seule instruction ou déclaration, mais ce n'est pas obligatoire.

Formation des mots

Un **mot** est formé d'une suite de caractères pris parmi les suivants : *lettres* majuscules et minuscules (anglaises, c'est-à-dire non accentuées), *chiffres*, *tiret* (sous la forme du caractère de soulignement _); le premier caractère d'un mot ne peut pas être un chiffre. Exemples : `while return n p puiss prix_unitaire Qte etat_1 Etat_2 __MIN__`

La longueur maximum d'un mot est de 31 caractères.[4]

En C, **une minuscule et une majuscule ne sont pas équivalentes** et interchangeables. (Les trois mots `resultat Resultat RESULTAT` sont différents et désigneraient trois objets distincts).

Les mots clés du langage sont tous en minuscules. Par tradition, on préfère écrire en minuscules.

Liste des mots réservés

Le programmeur ne peut pas redéfinir les mots clés du langage; ceux-ci sont "réservés".

auto	double	int	struct
break	else	long	switch
case	enum	register	typedef
char	extern	return	union
const	float	short	unsigned
continue	for	signed	void
default	goto	sizeof	volatile
do	if	static	while

[4] En réalité, un mot peut être plus long, mais le compilateur n'est obligé de considérer que les 31 premiers caractères.

4.2. Variante

En vertu des possibilités syntaxiques du langage, des variantes sont possibles :

```
float puiss(float n,int p)    /* élever n à la puissance p */
{
   float resultat=1.0;
   for(;p>0;--p) resultat *= n;
   for(;p<0;++p) resultat /= n;
   return (resultat);
}
```

Cette variante, remarquablement compacte, a été obtenue en utilisant diverses possibilités du langage C, dont certains opérateurs gadgets :

- si θ est un opérateur arithmétique `+ - * /` , l'expression `x θ= y` équivaut à `x = x θ y`
- l'expression `++ x` équivaut à `x += 1` l'expression `-- x` équivaut à `x -= 1`

- la déclaration de la variable *resultat* comporte son initialisation à la valeur *1*

- lorsque la boucle de répétition `while` est contrôlée par un compteur, elle épouse ce schéma :

```
init; /* initialisation du compteur (sauf si sa valeur initiale est déjà acquise) */
while (cond)              /* condition de poursuite de l'exécution */
{
 oper;                    /* opérations */
  incr;                   /* incrémentation du compteur */
}
```

cette construction peut être remplacée par : `for (init;cond;incr) {oper;}`

de plus, dans le cas présent :

- les accolades `{}` sont supprimées autour du bloc *oper*, parce qu'il comporte une seule instruction
- le traitement *init* d'initialisation du compteur est vide

4.3. Etape 2.2 : ajouter des opérations d'échange d'information

Pour créer un programme apte à exécuter cette fonction *puiss*, nous devons ajouter des opérations visant à :

- déterminer la *valeur des arguments* N et P
  (par exemple, en demandant de les introduire au clavier);
- délivrer le *résultat* (par exemple, en l'affichant à l'écran);
- éventuellement, tester la *validité* des arguments introduits
  (message d'erreur à l'écran et redemander l'introduction).

Ces opérations assurent la **communication**, l'**échange** de l'information. Il existe diverses sortes d'échanges :

- entre l'ordinateur et l'opérateur humain – via des **terminaux** (écran, clavier, imprimante);
- entre programmes se déroulant successivement – par le truchement de **fichiers**, collections de données conservées sur des supports magnétiques (disques, disquettes, bandes);
- entre programmes s'exécutant simultanément – à travers des **réseaux** de (télé)communication.

Nous programmerons ces opérations (échanges avec l'utilisateur) dans une autre fonction :

```
void main (void)  /* void ("vide") = ni résultat, ni paramètres */
/* test de la fonction puiss()
   - appelle les fonctions printf() : affichage à l'écran
                           scanf() :  lecture au clavier  */
{
   float r;
   int e;
   printf("\n introduisez la racine [réelle] : ");scanf("%g",&r);
   printf("\n introduisez l'exposant [entier] : ");scanf("%d",&e);
   printf("\n %g exp %d = %g \n",r,e,puiss(r,e));
}
```

Dans la définition d'une fonction, le mot clé ***void*** ("vide") signifie soit que cette fonction ne reçoit pas de paramètres, soit qu'elle ne renvoie pas de résultat.

Un programme C est un ensemble de fonctions.

Un programme C est un ensemble de fonctions rédigées par le programmeur – ex.: *main(), puiss()* – ou conservées dans des collections ou **bibliothèques**, standards (c'est-à-dire fournies avec le compilateur du langage) ou non – ex.: *printf(), scanf()*.[5]

Parmi les fonctions d'un programme C, il doit exister une et une seule fonction **main()**. Mis à part un cas particulier qui sera décrit dans un chapitre ultérieur, cette fonction ne reçoit *pas de paramètres*. Où qu'elle soit rédigée, elle est la première à s'exécuter au démarrage du programme. La fin d'exécution de la fonction *main()* provoque la terminaison de l'exécution du programme complet.

Présentation des fonctions standards de dialogue avec l'opérateur

(Description partielle)

La fonction *printf() "print with format"* affiche un texte à l'écran, la fonction *scanf() "scan with format"* lit la réponse introduite au clavier. Ces deux fonctions reçoivent un ou plusieurs paramètres; le premier est une chaîne de caractères (entre " ") de contrôle, qui décrit le format du texte affiché ou lu; les paramètres suivants contiennent les valeurs véhiculées par l'échange.

printf() – "print with format"
• *paramètres*: chaîne de contrôle puis de 0 à n *valeurs* (constantes, variables, fonctions, expressions)
• *retour* : néant
La chaîne de contrôle contient du texte, des commandes de mise en page (**`\n`**) et des codes (**`%`**) définissant le format sous lequel doivent être affichées les valeurs des paramètres suivants.

[5] Lorsqu'on cite le nom d'une fonction, on fait habituellement suivre ce nom d'une paire de parenthèses. Cette convention permet de distinguer les noms de fonctions et de variables.

scanf() – "scan with format"
• *paramètres* : chaîne de contrôle puis de 1 à n *valeurs d'adresses* (syntaxe : *&nom*) – voir ci-après
• *retour* : néant
La chaîne de contrôle contient des codes (`%`) définissant le format sous lequel les valeurs doivent être introduites au clavier.

Le premier code `%` définit le format de la première valeur, le deuxième code `%` définit le format de la deuxième valeur, et ainsi de suite.

```
printf("\n %g exp %d = %g \n",r,e,puiss(r,e));
            1-----------------1
                   2------------2
                        3-----------3
```

formats	%c	*one* character	*ex.:*	a
principaux	%d	decimal integer		-237
	%e	réel en notation scientifique		45e-2
	%f	réel en point flottant		0.45
	%g	réel dans le format le plus adapté : %e ou %f		
	%s	character string		Liège
	%%	représentation du caractère % (ceci n'est pas un code de format)		

commandes	\a	alert	signal sonore
principales	\b	backspace	recule d'une position sur la ligne courante
	\f	form feed	va au début de la page suivante
	\n	new line	va au début de la ligne suivante
	\r	carriage return	retourne à la 1ère position de la ligne courante
	\t	horizontal tab	avance à la position du tabulateur suivant sur la ligne
	\v	vertical tab	avance à la ligne marquée d'une tabulation verticale
	\0	nul	caractère "nul" (marque de fin de texte)
	\\		\
	\'		' *caractères ayant une signification*
	\"		" *spéciale dans le langage C*
	\?		?

Appel de fonction et passation des paramètres

La fonction *main()* demande l'exécution des fonctions *puiss()*, *printf()* et *scanf()*; on dit qu'elle **appelle** ces fonctions. En C, un appel de fonction s'écrit de la manière suivante :

```
nom_de_fonction (liste_des_paramètres_effectifs)
```

Chaque **paramètre effectif** est une expression (constante, variable, expression arithmétique, appel de fonction ...) fournissant une ***valeur***.

Ceci produit, dans l'ordre, les effets suivants :

1) passation des valeurs des paramètres effectifs à la fonction appelée, qui les range dans les paramètres formels dont elle contient la déclaration :

> une *copie* de la valeur du premier paramètre effectif est rangée dans le premier paramètre formel, une *copie* de la valeur du deuxième paramètre effectif est rangée dans le deuxième paramètre formel, etc. (en C, les paramètres sont "***passés par valeur***")
>
> ```
> appel :                  puiss (      r,      e)   paramètres effectifs
>                                       ↓       ↓
> déclaration :    float puiss (float n, int p)   paramètres formels
> ```
>
> les paramètres effectifs et formels se correspondent par leur position dans les listes de paramètres, et non pas par leurs noms (un paramètre effectif donné sous la forme d'une expression arithmétique ne possède pas de nom !);

2) exécution des opérations de la fonction appelée;
3) si la fonction appelée "retourne" un résultat, réception de cette valeur par la fonction appelante;
4) poursuite de l'exécution de la fonction appelante.

Puisqu'une fonction appelée travaille sur des copies de ses paramètres, elle peut modifier la valeur de ses paramètres formels, ce qui laisse inchangée la valeur des paramètres effectifs dans la fonction appelante. (La fonction *puiss()* peut donc utiliser son paramètre formel p pour compter les itérations du traitement.)

Une fonction comme *scanf()* doit modifier la valeur des paramètres effectifs de la fonction appelante : elle doit ranger les valeurs reçues du clavier dans les variables de la fonction appelante. On doit donc lui *passer sous la forme de valeurs les adresses* de ces variables. On emploie pour cela l'opérateur & : l'expression `&var` transforme en valeur l'adresse de la variable de nom `var` .

main				**scanf**		
nom	*adr.*	*valeur*		*valeur*	*adr.*	*nom*
	16				01	
r	17	2.5	&r →	17	02	?
e	18	- 3	&e →	18	03	?
	19				04	

> On appelle **pointeur** une donnée qui, en guise de valeur, contient l'adresse d'une autre. (Les paramètres formels de la fonction *scanf()* sont des pointeurs.)

4.4. Etape 2.3 : organiser les "bibliothèques" de programmes

Répartition du texte en un ou plusieurs fichiers

Le texte d'un programme peut être réparti dans un ou plusieurs **fichiers** *nnnnnn.c* .

Répartition en un seul fichier :

puiss.c

```
float puiss(float n,int p)
/* élever n (réel) à la puissance p (entier signé) */
{
   float resultat=1.0;
   for(;p>0;--p) resultat *= n;
   for(;p<0;++p) resultat /= n;
   return (resultat);
}

void main (void)  /* test de la fonction puiss() */
{
   float r;
   int e;
   printf("\n introduisez la racine [réelle] : ");scanf("%g",&r);
   printf("\n introduisez l'exposant [entier] : ");scanf("%d",&e);
   printf("\n %g exp %d = %g \n",r,e,puiss(r,e));
}
```

REGLE IMPORTANTE. En principe, **toute fonction doit être déclarée avant d'être appelée par une autre** (dans notre exemple, *puiss()* est déclarée avant *main()*); ceci permet au compilateur, lorsqu'il analyse le texte d'une fonction appelante, de connaître le type et le mode de représentation des paramètres à fournir et du résultat à récupérer. A défaut, comme pour *printf()* ou *scanf()* dans notre exemple, le compilateur fait des présomptions.

Dans la méthode illustrée ci-dessus, les deux fonctions sont déclarées *et définies* dans le même fichier.

Répartition en plusieurs fichiers :

puiss.c

```
float puiss(float n,int p)
/* élever n (réel) à la puissance p (entier signé) */
{
   float resultat=1.0;
   for(;p>0;--p) resultat *= n;
   for(;p<0;++p) resultat /= n;
   return (resultat);
}
```

test.c

```
float puiss(float n,int p);  /* "prototype" de la fonct. appelée */

void main (void)  /* test de la fonction puiss() */
{
   float r;
   int e;
   printf("\n introduisez la racine [réelle] : ");scanf("%g",&r);
   printf("\n introduisez l'exposant [entier] : ");scanf("%d",&e);
   printf("\n %g exp %d = %g \n",r,e,puiss(r,e));
}
```

Dans la méthode illustrée ci-dessus, la fonction *puiss()* est déclarée et définie dans le fichier *puiss.c*, et un rappel de sa déclaration est inséré dans le fichier *test.c*. Ce rappel de déclaration est appelé le **prototype** de la fonction.

Utilisation de fichiers d'en-tête :

Troisième méthode : si le fichier *puiss.c* a été créé par un autre programmeur, ce dernier peut avoir écrit lui-même le prototype à utiliser et l'avoir stocké dans un **fichier d'en-tête** *nnnnnn.h* (*.h* = "header").

puiss.h

```
float puiss(float n, int p);        /* prototype */
```

test.c

```
#include "puiss.h"      /* fichier des déclarations nécessaires
                                à l'utilisation de puiss() */
#include <stdio.h>      /* fichier des déclarations <standards>
                           nécessaires à l'utilisation des fonct.
                           "standard input/output" */
void main (void)  /* test de la fonction puiss() */
{
   float r;
   int e;
   printf("\n introduisez la racine [réelle] : ");scanf("%g",&r);
   printf("\n introduisez l'exposant [entier] : ");scanf("%d",&e);
   printf("\n %g exp %d = %g \n",r,e,puiss(r,e));
}
```

Création du programme exécutable

Pour devenir un programme exécutable, le programme rédigé (qu'on appelle ***source***) doit subir une série de transformations, opérées par des programmes "utilitaires" préexistants.

Un programme **éditeur de textes** est utilisé pour enregistrer le texte source sur un support magnétique (disque ou disquette).

Le **compilateur** est la pièce centrale. Ce programme

- analyse un fichier source,
- crée un fichier "***objet***",
- produit une liste de contrôle (diagnostic des erreurs, etc).

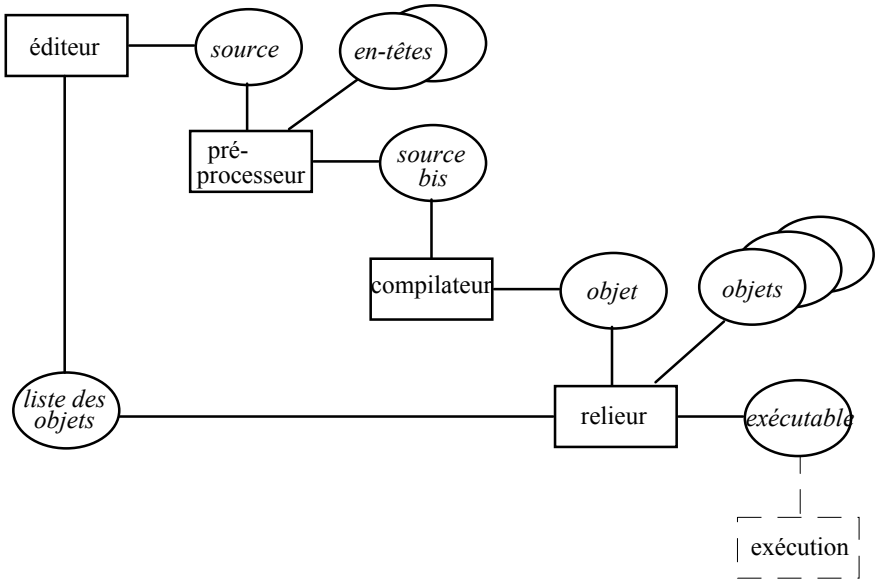
Le **pré-processeur** prépare le texte à donner au compilateur. Ce programme

- supprime les commentaires **/* */** ,
- inclut les fichiers d'en-tête **.h** ,
- exécute les autres directives **#** .

Le **relieur** ou **éditeur de liens** combine en un seul texte ***exécutable*** les différents fichiers objets nécessaires. La liste des fichiers objets à rassembler est fournie par un fichier créé au moyen de l'éditeur de textes. Exemple pour le système Borland/Turbo C :

puiss.prj

`puiss.c` `test.c (puiss.h)`	chaque fois qu'un de ces fichiers est modifié, l'éditeur de liens reconstruit le programme exécutable la mention des fichiers `.h` entre parenthèses peut être omise



Remarque. Les programmes pré-processeur, compilateur et relieur sont souvent activés en une seule commande.

5. Un autre exemple

5.1. Algorithmes[6]

Calculer le plus grand commun diviseur (PGCD) de deux nombres A et B et leur plus petit commun multiple (PPCM).

Les algorithmes de résolution se fondent sur les **relations mathématiques** suivantes :

- type des arguments : les deux nombres A et B sont des entiers strictement positifs

- PGCD : si A = B, le PGCD est A ou B
  le PGCD de deux nombres est inchangé
  si l'on substitue au plus grand la différence des deux

  soit : si A = B, PGCD (A, B) = A (ou B)
  si A < B, PGCD (A, B) = PGCD (A, B – A)
  si A > B, PGCD (A, B) = PGCD (A – B, B)

[6] Exemple tiré de N. WIRTH : *Programmer en MODULA-2*; Presses Polytechniques Romandes; Lausanne, 1984.

• PPCM : A x B = PGCD (A, B) x PPCM (A, B)

Algorithmes

• **pgcd (a, b)** : 1) si a et b sont égaux, le résultat est a (ou b)
2) sinon, prendre le pgcd du plus petit des deux nombres et de leur différence
c'est-à-dire réexécuter l'algorithme sur ces nouvelles données
(la différence atteindra finalement 0, et l'on finira par tomber dans le cas 1
ex.: pgcd(8,20) ← pgcd(8,12) ← pgcd(8,4) ← *pgcd(4,4)*)

METHODE :
```
SI a = b
ALORS résultat ← a
SINON  DEBUT SI a < b
                    ALORS résultat ← pgcd (a, b – a)
                    FIN-SI
                    SI b < a
                    ALORS résultat ← pgcd (b, a – b)
                    FIN-SI
             FIN
FIN-SI
```

Cet algorithme utilise un nouveau mécanisme de composition d'opérations : la **récursivité**. On qualifie de récursif un algorithme qui se rappelle lui-même (dans l'exemple, lorsque a ≠ b).

• **ppcm (a,b)** : *METHODE :* résultat ← (a x b) / pgcd (a, b)

5.2. Programme

pgcdppcm.c

```
int pgcd (int a, int b)
{
   if (a == b) return a;
   if (a < b) return pgcd (a, b - a);
   if (a > b) return pgcd (b, a - b);
}

int ppcm (int a, int b)
{
   return ( (a * b) / pgcd (a, b) );
}
```

test.c

```
#include <stdio.h>
int pgcd(int a, int b);
int ppcm(int a, int b);

void main (void)        /* test des fonctions pgcd() et ppcm() */
{
   int a; int b;
   printf ("\n introduisez deux nombres entiers positifs ");
   scanf ("%d%d", &a, &b);
   printf ("\n pgcd = %d  ppcm = %d \n", pgcd (a, b), ppcm (a, b));
}
```

Variante non récursive de la fonction pgcd().

Le texte fait moins transparaître le raisonnement, récursif, qui a conduit à la solution.

```
int pgcd (int a, int b)
{
   while (a != b)       /* !=  : opérateur "non égal" */
   {
      if (a < b) b = b - a;         /* SI ... */
      else                          /* SINON  */
      if (b < a) a = a - b;
   }
   return a;
}
```

5.3. Le concept de module

Un **module** est un texte séparé (source ou objet) dans lequel sont regroupées une partie des fonctions d'un programme. Par exemple, chacun des fichiers *pgcdppcm.c* et *test.c* rédigés plus haut contient un module source.

Les regroupements en modules ne sont évidemment pas aléatoires ! Les fonctions rassemblées dans un même module ont entre elles divers liens de *parenté* (similitude ou complémentarité) et d'*usage*.

Exemple.

> Les deux fonctions *pgcd()* et *ppcm()* rassemblées dans le module *pgcdppcm* sont complémentaires, elles manipulent les mêmes données, et l'une (*ppcm*) a besoin de l'autre (*pgcd*). Elles ont des applications dans les mêmes domaines de l'activité humaine et sont mises à la disposition de tous les programmeurs.
>
> Le module *test* contient des fonctions utiles, provisoirement, au seul programmeur occupé à la mise au point du module *pgcdppcm*.

6. *Supplément.* Note sur la documentation des fonctions

Lorsqu'un programmeur crée des fonctions qu'il désire mettre à la disposition d'autres programmeurs (et c'est toujours le cas, puisque tout programmeur aura des successeurs ...), il doit fournir aux utilisateurs futurs de ces fonctions une documentation (un mode d'emploi) suffisante et correcte; en même temps, il souhaite protéger les textes qu'il a programmés de toute "intrusion" (indiscrétion ou modification) malencontreuse.

La documentation d'une fonction doit fournir les renseignements suivants :

- le nom de la fonction;
- la liste ordonnée des paramètres;
- le type des paramètres et le type du résultat,
  à compléter par la liste des valeurs exclues;
- un commentaire précisant les autres pré-conditions éventuelles;
- si un paramètre ou le résultat possède des valeurs codées bien définies,
  un commentaire énumérant ces valeurs avec leur signification;
- un commentaire expliquant de manière compréhensible
  la transformation opérée sur les paramètres.

A l'exception des commentaires, ces indications composent le *prototype* de la fonction.

La documentation des fonctions est donc faite dans les ***fichiers d'en-tête*** `nnnnnn.h` ; ce sont les seuls fichiers auxquels les programmeurs utilisateurs auront accès.

Pour éviter une dispersion excessive de la documentation, on groupera dans un même fichier d'en-tête la description de toutes les fonctions d'un même module ou d'une même "famille". C'est ainsi qu'ont procédé les créateurs des fonctions de la bibliothèque standard associée au langage C.

Exemple :

pgcdppcm.h

```
/* === pgcdppcm.h === */
/* module : fonctions mathématiques utilitaires */
/*--->
   version 2 - 27/09/93 - auteur : A.C.
   version 1 - 10/09/93 - auteur : A.C.
<---*/

int pgcd(int a, int b);
/*
   calcul du plus grand commun diviseur de deux nombres entiers
   algorithme récursif
   - tous les nombres sont strictement > 0
*/

int ppcm(int a, int b);
/*
    calcul du plus petit commun multiple de deux nombres entiers
    en utilisant la formule (a*b) == pgcd(a,b)*ppcm(a,b)
    - tous les nombres sont strictement > 0
*/
```

Exercices

1.

Tester les programmes donnés en exemple dans le texte du chapitre.
Essayer les différents modes de répartition en fichiers sources.

2.

Telle qu'elle est programmée dans les exemples, la fonction *main()* permet de tester chaque fonction de calcul pour un seul couple de valeurs. Procéder aux adaptations nécessaires pour pouvoir répéter le test un nombre indéterminé de fois, jusqu'à ce que le premier nombre du couple introduit soit 0.

L'algorithme est le suivant :

```
demander un premier couple de valeurs
TANT QUE le premier nombre introduit est différent de 0
EXECUTER      appeler la fonction à tester
              demander un nouveau couple de valeurs
FIN-TANT
```

3.

Sur le modèle de la fonction *pgcd()*, rédiger une fonction calculant la factorielle *n!* d'un nombre entier positif. Ecrire les deux formes – récursive et non récursive – de la fonction.

Définition intuitive :

```
0! = 1
1! = 1 x 1 = 1 x 0!
2! = 2 x 1 = 2 x 1!
3! = 3 x 2 x 1 = 3 x 2!
4! = 4 x 3 x 2 x 1 = 4 x 3!
.....
```

Chapitre 2. Les valeurs

1. Introduction : principes de représentation de l'information

1.1. Le système de numération binaire

L'ordinateur manipule des représentations électro-magnétiques de l'information. Ces représentations utilisent, en guise d'équivalents des chiffres binaires 0 et 1, les deux états physiques de matériaux qualifiés de "bi-stables". Un chiffre binaire a reçu le nom de **bit** ("BInary digIT", en anglais).

Ces chiffres sont groupés en "mots" de différentes tailles. Ces groupements de bits forment des **nombres en base 2** ou **nombres binaires**, constitués selon les mêmes principes que ceux du système de numération décimale (en base 10) :

- soit b le nombre de chiffres (signes graphiques) disponibles, symbolisant les valeurs 0 à $b-1$;
- soit un groupe de n chiffres dont, de droite à gauche, les positions sont numérotées de 0 à $n-1$;
- le chiffre c à la position p représente la valeur $c \times b^p$;
- on dit que b^p est le "poids" de la position p
- les positions de droite ont des poids faibles et les positions de gauche, des poids forts;
- la valeur du nombre égale la somme des valeurs représentées par les différents chiffres;
- sur n positions peuvent être codés b^n nombres, dont les valeurs s'échelonnent de 0 à $b^n - 1$.

Exemple : $1001_2 = 09_{10}$

$= 1 \times 2^0 = 9 \times 10^0$
$+ 0 \times 2^1 + 0 \times 10^1$
$+ 0 \times 2^2$
$+ 1 \times 2^3$

Il est malaisé de lire un nombre binaire; aussi les informaticiens (pas l'ordinateur !) emploient-ils d'autres bases que la base 2 :

• dans le système **octal**, les nombres s'écrivent au moyen des 8 chiffres **0** à **7** – chacun de ces chiffres synthétise un groupe de 3 bits;

exemples : $:010:110:_2 = 26_8$ $:000:111:_2 = 07_8$

• dans le système **hexadécimal**, les nombres s'écrivent au moyen des 16 chiffres **0** à **9** et **A** à **F** (pour les valeurs 10 à 15) – chacun de ces chiffres synthétise un groupe de 4 bits;

exemples : $:0101:1100:_2 = 5C_{16}$ $:0000:1111:_2 = 0F_{16}$

1.2. Taille des représentations binaires

L'unité centrale d'un ordinateur est principalement composée de deux organes :

- le ***processeur*** ou ensemble des circuits logiques opérateurs;
- la ***mémoire centrale***, où sont stockées les données (représentations des informations) pour la durée de leur traitement.

La *mémoire centrale* d'un ordinateur est logiquement découpée en **mots**, groupes de bits d'une longueur déterminée. Les mots sont numérotés; le numéro d'un mot est appelé l'***adresse*** de ce mot.

Aujourd'hui, pratiquement tous les ordinateurs adoptent la même taille pour le mot de mémoire : l'**octet**, mot de 8 bits. Dans un octet peuvent être codés 2^8 ou 256 nombres binaires.

Le *processeur* comporte un certain nombre de **registres** via lesquels s'opère l'échange de données avec la mémoire centrale. La taille d'un registre est un multiple de celle d'un octet, et elle est suffisante pour contenir le nombre qui représente l'adresse d'un mot de mémoire quelconque.

Dans la plupart des ordinateurs actuels, la taille d'un registre est de 32 bits (4 octets) ou 16 bits (2 octets). Un registre de 32 bits permet d'adresser 2^{32} mots de mémoire, un registre de 16 bits permet d'adresser 2^{16} mots de mémoire.

La taille d'un registre d'adressage est la longueur de mot privilégiée par le langage C.

1.3. Interprétation des représentations binaires

Selon le contexte où elle est utilisée, ***une même représentation binaire peut faire l'objet de différentes interprétations*** :

- **nombre entier non signé** (positif) :

  tous les bits du mot sont interprétés comme étant les chiffres du nombre.

- **nombre entier signé** :

  le bit de poids le plus fort est interprété comme le signe du nombre, et la valeur du nombre est codée dans les bits de droite :

  – si le bit de signe est 0, le nombre est positif
  et sa valeur est codée sur les autres bits à la manière d'un entier non signé
  (sur `n` bits, sans compter le signe, la plus grande valeur représentable est $+2^n-1$);

  – si le bit de signe est 1, le nombre est négatif
  et sa valeur est codée sur les autres bits comme le complément à 2 de sa valeur absolue
  (sur `n` bits, sans compter le signe, la plus petite valeur représentable est -2^n);

la configuration binaire d'un nombre négatif est obtenue de la manière suivante :

- prendre en binaire la valeur absolue (ex.: +5 `00000101`)
- bit à bit, prendre le complément à 1 (ex.: `11111010`)
- ajouter 1 (ex.: −5 `11111011`)

cette représentation binaire se justifie comme ceci :
il suffit d'y ajouter la valeur absolue (nombre opposé) pour obtenir 0

ex.:	reports:	*11111111*	
	−5	11111011	*"1 + 1 = 10_2*
	+5	00000101	*j'écris 0 et je reporte 1"*
	=	00000000	

- **nombre réel en virgule flottante** :

l'interprétation distingue dans le mot deux sous-groupes de bits,
l'un représentant la *mantisse* entière et l'autre, l'*exposant*;
chacune des deux parties est signée de la même manière qu'un nombre entier;

soit b la base du système de numération;
soit m la mantisse, e l'exposant; la valeur du nombre est $m \times b^e$

mantisse	exposant	base = 10
exemples :		
+ 453	− 1	+45.3
− 12	+ 6	−12000000
+ 5	− 3	+0.005

- la taille de la mantisse détermine la *précision* de la représentation,
  c'est-à-dire le nombre de chiffres significatifs représentables;
- l'exposant est un facteur d'échelle décimale qui détermine la position de la virgule.[7]

- **caractère** :

dans un octet, il est possible de coder 2^8 ou 256 représentations binaires; lorsque ces représentations sont transférées sur un appareil de visualisation – écran ou imprimante –, elles se voient conventionnellement correspondre 256 caractères : chiffres décimaux, lettres minuscules et majuscules, autres signes.

Le concept d'alphabet

En informatique, un **alphabet** est un *ensemble de caractères*, ensemble muni d'un ordre appelé *séquence de classement*, ce qui rend les caractères comparables : A < B < C ... La position d'un caractère dans l'alphabet est indiquée par la valeur numérique à laquelle il correspond.

[7] On a décrit ici le mécanisme de virgule flottante tel qu'il est perçu par un programmeur C (voir ci-après : *Les constantes littérales*). En réalité, la base *b = 2* et la mantisse est "normalisée" : $1/b \le m < 1$; de plus, les positions relatives de la mantisse et de l'exposant sont inversées.

Fondamentalement, un "caractère" (élément d'un alphabet) est un nombre binaire représentable en mémoire centrale de l'ordinateur. Tous les éléments d'un alphabet ne sont pas nécessairement représentables sur les autres supports; en particulier, certains "caractères" n'ont pas de représentation graphique. Ils servent de signaux pour la commande des terminaux (fin de ligne, signal sonore ...), des imprimantes (saut de ligne ou de page ...), des télétransmissions (annulation, fin de texte ...).

- **valeur logique ou booléenne** :

  un nombre binaire peut être interprété comme figurant une des deux valeurs logiques *vrai* ou *faux*;

  pour le langage C,

  - le nombre 0 représente la valeur *faux*,
  - tout autre nombre représente la valeur *vrai*.

- **adresse** de mémoire :

  un nombre binaire peut être compris comme l'adresse (entière non signée) d'un mot de mémoire.

- **instruction** :

  un groupe de bits peut être interprété comme formant une instruction, dans laquelle on distingue différents champs : un *code d'opération* et des *adresses* d'opérandes.

1.4. Exercices

Imprimer la partie graphique de l'alphabet ASCII

Un grand nombre d'ordinateurs actuels emploient l'alphabet standard ASCII ("American Standard Code for Information Interchange").[8] Les caractères ayant un équivalent graphique y occupent les positions 32 à 126.

On demande d'imprimer la partie graphique de cet alphabet, en présentant chaque caractère sous trois formes : valeur numérique décimale, valeur numérique hexadécimale, symbole graphique.

Algorithme : impression de l'alphabet (partie graphique) : 8 caractères par ligne

```
DEBUT i ← 32
        TANT QUE i < 127
        EXECUTER    DEBUT SI reste de i / 8 = 0
                          ALORS aller à la ligne
                          FIN-SI
                          imprimer les trois formes du caractère correspondant
                          i ← i + 1
                    FIN
        FIN-TANT
FIN
```

[8] Cet alphabet comporte 128 caractères; il occupe les positions 0 à 127 de l'alphabet de l'ordinateur, tandis que le contenu des positions 128 à 255 diffère d'un ordinateur à l'autre.

Pour l'affichage des valeurs, la fonction **printf()** doit être appelée avec d'autres formats que ceux définis au chapitre 1.

printf() – "print with format"

formats	%c	*one* character
	%..d	decimal integer
	%..x	hexadecimal integer

– dans un code de format, les .. suivant immédiatement le signe % signalent l'endroit où peut être indiqué le nombre de chiffres à afficher;
– si ce nombre commence par 0, les positions non significatives sont mises à zéro plutôt qu'à blanc.

```
#include <stdio.h>

void alphabet(void)

/* impression de l'alphabet ASCII
   (partie graphique)
   8 caractères par ligne
   formats :
   décimal, hexadécimal, symbole  */

{
   int i;
   i = 32;
   while (i < 127)
   {
      if (i%8 == 0) printf ("\n");
      printf ("%03d %02x %c  ",i,i,i);
      ++i;
   }

   return;
}
```

void :
la fonction ne reçoit pas de paramètres et ne renvoie pas de valeur résultat,
ce que signifie le pseudo-type ***void*** (vide)

% opérateur *modulo* (reste de la division par)

return : retour sans renvoi de valeur
– lorsque, comme ici, cette instruction est la dernière du corps de la fonction, elle peut être omise

Simuler le mécanisme de la virgule flottante

En s'inspirant de la fonction *puiss()* présentée au chapitre 1, écrire une fonction qui, recevant en paramètres une mantisse et un exposant, calcule et renvoie la valeur réelle représentée.

```
float puiss(float n,int p);
      /* rappel de la déclaration de la fonction puiss() */

float reel(float m,int e)
/* mantisse,exposant -> calculer la valeur réelle */
{   return (m * puiss(10,e));   }
```

2. Les types de données scalaires

Le langage C définit un certain nombre de types de données. On verra que, sur le base de ces types prédéfinis, le programmeur peut en construire d'autres.

Les types de données prédéfinis sont les suivants :

types entiers signés	*taille standard*	*taille habituelle*	*valeurs habituelles*
signed char (*"character"*)	1 octet	1 octet = 8 bits	- 128 ... + 127
signed short int	2 octets	2 octets = 16 bits	- 32 768 ... + 32 767
signed int (*"integer"*)	1 registre adresse	2 ou 4 octets	
signed long int	4 octets	4 octets = 32 bits	- 2 147 483 648 .. + 2 147 483 647

Le qualificatif **signed** peut être omis.

types entiers non signés[9]	*taille standard*	*taille habituelle*	*valeurs habituelles*
unsigned char	1 octet	1 octet = 8 bits	0 ... 255
unsigned short int	2 octets	2 octets = 16 bits	0 ... 65 535
unsigned int	1 registre adresse	2 ou 4 octets	
unsigned long int	4 octets	4 octets = 32 bits	0 ... 4 294 967 295

types réels	*taille standard*	*taille habituelle*	*valeurs habituelles*
float	simple précision	4 octets = 32 bits	mantisse : ± 6 chiffres significatifs
double	double précision	8 octets = 64 bits	mantisse : ± 12 chiffres significatifs
long double	précision étendue	16 octets = 128 bits	

REMARQUES

Les *tailles standards* indiquées ci-dessus sont les tailles minimales imposées par la norme ANSI C.

Dans tous les cas, les relations suivantes doivent être respectées par les compilateurs :

- taille *char* < taille *short int* ≤ taille *int* ≤ taille *long int*
- taille *float* ≤ taille *double* ≤ taille *long double*

La taille du type **int** n'est pas uniforme; elle varie d'un ordinateur à l'autre. Ceci peut poser de sérieux problèmes lorsque l'on veut "porter" un programme d'un ordinateur sur un autre.

Le type **int** est le *type privilégié* du langage C. ***Lorsqu'elle est utilisée à l'intérieur d'une expression (arithmétique, par exemple), une donnée d'un type plus court (`char` ou `short`) est automatiquement transposée dans le type `int`.***

L'opérateur sizeof()

L'opérateur **sizeof()** renvoie, sous la forme d'un entier, la taille d'un objet en nombre d'octets. Son paramètre est soit un nom de ***type***, soit une ***expression*** (la plus simple étant un nom de variable).

[9] La plupart des langages de programmation n'ont pas de types **unsigned**. Le langage C vise ici à offrir des opérations équivalentes aux opérations des langages assembleurs.

Les relations suivantes sont donc vraies :

```
sizeof(char) < sizeof(short int) ≤ sizeof(int) ≤ sizeof(long int)
sizeof(float) ≤ sizeof(double) ≤ sizeof(long double)
```

3. Les constantes littérales

Une **constante** est une zone de mémoire dont le contenu (la valeur) ne peut pas être modifié pendant l'exécution du programme. Le plus souvent, pour référencer une constante, on en écrit directement la valeur – on dit qu'on utilise des constantes *littérales*. La valeur des constantes littérales est connue dès le stade de la compilation.

3.1 Constantes scalaires

1) Une constante **entière** peut prendre trois formes :

- nombre ***décimal*** : suite de chiffres 0 à 9 *ne commençant pas par 0 (zéro)*;
- nombre ***octal*** : suite de chiffres 0 à 7 *commençant par 0 (zéro)*;
- nombre ***hexadécimal*** : suite de chiffres 0 à 9 et A à F *commençant par 0X (zéro X)*.

Par défaut, le type d'une constante entière est, parmi les suivants, le premier qui puisse contenir sa valeur : **`signed int`**, **`signed long int`**, **`unsigned long int`**. Si les chiffres sont immédiatement suivis de la lettre **U**, le type est **`unsigned`**; enfin, le suffixe **L** force le type à **`long int`**.

exemples : toutes les constantes ci-dessous expriment la même valeur :

décimal	*octal*	*hexadécimal*	*type*
`159`	`0237`	`0x9F`	signed int
`159U`	`0237u`	`0x9Fu`	unsigned int
`159L`	`0237l`	`0X9fL`	signed long int
`159UL`	`0237ul`	`0x9Ful`	unsigned long int

2) Une constante **réelle** prend la forme suivante :

- *mantisse* [**`E`** [*signe*] *exposant*]
- la mantisse est obligatoire, l'exposant est facultatif et peut être signé par + ou – ;
- la mantisse prend la forme suivante [*partie entière*] [**`.`** *partie décimale*]
- la constante ne peut pas être formée de la seule partie entière de la mantisse;
- chacune des trois parties (entière, décimale, exposant) est une suite de chiffres décimaux;

soit m la mantisse, e l'exposant; la valeur de la constante est $m \times 10^e$

Par défaut, une constante réelle est de type **`double`**. Le suffixe **F** ou **L** indique, respectivement, qu'elle est de type **`float`** ou **`long double`**.

exemples : toutes les constantes ci-dessous expriment la même valeur :

```
4.5   45e-1   .45e+1   .45E1   4.5E0     4.5e0f   45e-1L
```

Remarques

Les lettres figurant dans une constante numérique peuvent indifféremment s'écrire en minuscules ou majuscules.

A strictement parler, une constante numérique n'est jamais signée mais, dans toutes les expressions où une telle constante peut être employée, il est permis de la faire précéder de l'*opérateur unaire* **−** (moins), qui rendra sa valeur négative.

3) La valeur d'une constante de type **`char`** s'exprime sous la forme d'**un *caractère entre apostrophes*** :

- **caractère représentable** : exemples : `'a'  'B'  '+'  '3'  ';'`
- **caractère non représentable** : exemples: `'\n'  '\0'  '\\'  '\xff'` (voir ci-dessous)

Remarque

Une constante caractère peut être remplacée par une constante entière qui en indique la valeur numérique.

Liste des caractères non représentables ("escape sequences")

\a	alert	signal sonore	
\b	backspace	recule d'une position sur la ligne courante	
\f	form feed	va au début de la page suivante	
\n	new line	va au début de la ligne suivante	
\r	carriage return	retourne à la 1ère position de la ligne courante	
\t	horizontal tab	avance à la position du tabulateur suivant sur la ligne	
\v	vertical tab	avance à la ligne marquée d'une tabulation verticale	
\\		\	
\'		'	*caractères ayant une signification*
\"		"	*spéciale dans le langage C*
\?		?	
\ suivi de 1 à 3 chiffres octaux			*valeur numérique*
\x suivi de 1 à 2 chiffres hexadécimaux			*d'un caractère quelconque*

3.2. Chaînes de caractères

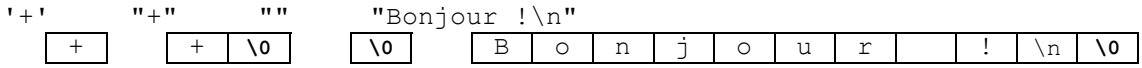
Le langage C offre une autre forme de constante littérale, qui ne représente pas une valeur scalaire : la **chaîne de caractères** ("character string") :

- suite de caractères, représentables ou non, placée ***entre guillemets***

exemples : `"Bonjour !\n"` `""` (= chaîne vide)

ATTENTION ! Ne pas confondre une *chaîne* de caractères, entre guillemets (ex.: *"+"*) et un *caractère*, entre apostrophes (ex.: *'+'*) ! Car ***ces deux sortes de constantes ne se représentent pas de la même manière dans la mémoire de l'ordinateur.***

Une chaîne de caractères est codée dans une suite d'octets contigus (dont chacun contient un caractère), suivis d'un octet contenant, en guise de **terminateur**, un caractère nul (nombre 0 binaire).



Remarque. Le caractère **** est utilisé pour couper un texte en fin de ligne. Autre méthode : deux chaînes de caractères consécutives en forment en réalité une seule. Exemples équivalents :

```
printf("Bonjour !\n");
```

```
printf("Bon\
jour !\n");
```

```
printf("Bon"
  "jour !\n");
```

4. Les variables scalaires *(première approche)*

Une **variable** est un zone de mémoire dont la valeur (le contenu) peut être modifiée par l'exécution du programme.

Une **variable scalaire** est un "mot" de mémoire contenant, à tout moment, une *valeur unique*, modifiable mais non décomposable. (D'autres variables, que nous qualifierons de *composites*, seront étudiées plus tard, dont le contenu se décompose en une *liste de valeurs*.)

4.1. Déclaration des variables

Une variable doit être *déclarée* avant d'être *utilisée*.

Déclarer une variable, c'est lui donner un nom *identificateur*, en même temps qu'on en précise les *attributs*. Le principal de ces attributs est le *type* de valeurs.[10]

Le format de base d'une déclaration de variable est le suivant :

type identificateur ;

Toute déclaration doit être clôturée par un point-virgule.

Remarque. Si, dans une déclaration, le mot `int` est précédé d'au moins un autre mot, il peut être omis.

Exemples :

```
unsigned int prix_unitaire;
int n;      ⇔    signed int n;
float taux_tva;
float surface;
float volume;
double distance_intersiderale;
short i;    ⇔    signed short int i;
unsigned no_facture;    ⇔    unsigned int no_facture;
long int montant_total;
char code_langue;
char ponctuation;
```

[10] Cf. supra. Les autres attributs des variables seront étudiés au chapitre 7.

4.2. Usage des identificateurs

Suivant le contexte où il est **utilisé**, l'identificateur déclaré pour une variable désigne tantôt la *valeur* de cette variable, tantôt son *adresse*, c'est-à-dire l'emplacement qu'elle occupe dans la mémoire de l'ordinateur.

> Soit une instruction d'affectation `i = i + 1`. Dans l'expression de droite, l'identificateur `i` désigne la valeur de la variable (avant exécution de l'affectation); dans l'expression de gauche, `i` désigne l'adresse de la variable résultat.

4.3. Initialisation d'une variable

La déclaration d'une variable peut comporter une clause d'**initialisation**. Cette clause détermine quelle valeur "initiale" est rangée dans la variable avant toute opération du programme manipulant cette variable.

Cette valeur initiale est définie par une constante ou une expression (notamment arithmétique) qui ne cite comme valeurs que des constantes.

Exemples : déclarations et initialisations :

```
int i = 0;
int j = - 1;
float n = 3.0 / 7.0;[11]
char homme = 'M'; char femme = 'F';
```

5. Déclaration des fonctions *(première approche)*

La définition d'une fonction comporte, dans l'ordre, deux parties : la *signature* et le *corps* de la fonction.

`float puiss(float n,int p)`
`/* élever n`
`à la puissance p */`
	A. **signature** de la fonction :
	type_du_résultat nom (paramètres_formels)

`{`
`float resultat;`
 	
`resultat=1;`
`while(p>0)`
`{`
`resultat=resultat*n;`
`p=p-1;`
`}`
`while(p<0)`
`{`
`resultat=resultat/n;`
`p=p+1;`
`}`
`return (resultat);`
`}`
	B. **corps** de la fonction – bloc entre **{ }**
	déclaration des **variables locales**, c'est-à-dire accessibles aux seules opérations à l'intérieur du bloc
	opérations ou **instructions**

[11] `3.0` et `7.0` sont des nombres réels; le résultat de leur division est un nombre réel. L'expression `3/7`, portant sur des nombres entiers, donnerait un résultat entier (zéro), qui serait ensuite transposé dans le format "floating point". Cf chapitre 4.

• La **signature** d'une fonction décrit (déclare) les données – paramètres et résultat – qu'elle échange avec toute fonction appelante. Ces données forment l'***interface*** de la fonction, c'est-à-dire sa partie visible aux autres fonctions.

• Le **corps** d'une fonction est un bloc de texte entre accolades, définissant la partie interne et invisible de la fonction : variables locales et instructions.

Dans la définition d'une fonction, la signature comporte les éléments suivants :

type_du_résultat *identificateur* **(** *déclaration des paramètres formels* **)**

Cette déclaration est immédiatement suivie du bloc **{ }** formant le corps de la fonction.

Exemples :
```
float puiss (float racine, int exposant)
     { ........ }
int pgcd(int a,int b)
     { ........ }
int ppcm (int a, int b)
     { ........ }
void main (void)
     { ........ }
```

• **type** : le type mentionné est celui de la valeur résultat de la fonction.

– le qualificatif ***int*** (entier) peut être laissé implicite;
– le pseudo-type ***void*** ("vide") doit être indiqué pour une fonction qui ne renvoie pas de résultat.

• **liste des paramètres** :

– la liste comporte la déclaration (*type identificateur*) de chaque paramètre;
– ces déclarations sont séparées par des virgules;
– l'expression ***(void)*** ("vide") signifie que la fonction ne reçoit aucun paramètre.

Rappel de définition

Une fonction A peut être appelée par une fonction B programmée dans un autre fichier source que celui où A elle-même est définie. Pour cela, le **prototype** (rappel de la signature) de la fonction A appelée doit être inséré, en guise de rappel de la définition, dans le fichier source contenant le texte de la fonction B appelante. Le prototype rappelé n'est évidemment pas suivi du corps de la fonction.

type *identificateur* **(** *déclaration des paramètres formels* **)** **;**

Le rappel de déclaration est suivi du terminateur **;**

• **paramètres** : dans le rappel de définition d'une fonction, la déclaration de chaque paramètre peut être réduite à la seule indication de son **type**; l'identificateur a une utilité purement documentaire. Soulignons que la mention d'un identificateur "bien choisi" améliore grandement la lisibilité du programme.

Exemples :
```
int pgcd(int,int);
int ppcm (int, int);
float puiss(float,int);    ou, mieux :
float puiss (float racine, int exposant);
```

6. Les pointeurs et les constantes adresses *(première approche)*

6.1. Définitions

• Soit une variable `v`. De même que l'expression `sizeof(v)` fournit la taille de `v`, l'expression **`&v`** fournit l'adresse de `v`. Puisque cette adresse est immuable et connue du compilateur, l'expression `&v` est une **constante, de type "adresse"**.

• Un **pointeur** est une variable scalaire qui, en guise de valeur, contient l'adresse d'une autre variable.

Soit un pointeur `p` contenant l'adresse d'une variable `v`;
on dit que "`p` *pointe sur* `v`" et que "`v` est *repéré* par `p`".

On peut affecter à un pointeur une valeur-adresse : `p = &v`

• La **déclaration** d'un pointeur a la forme d'une déclaration de variable ordinaire, à ceci près que :

– le type indiqué est celui de l'objet repéré par le pointeur
(le pointeur ne pourra jamais pointer sur un objet d'un autre type);
– l'identificateur attribué au pointeur est préfixé par **`*`**.

exemples :

```
short int v=0;     /* variable de type short int */
short int * p=&v;
      /* pointeur sur un objet de type short int
         initialisé pour pointer sur l'objet v */
short int * q;     /* pointeur non initialisé */
```

	id	*adr*	*val*
`short int v = 0,`	`v`	16	0
`* p = &v,`	`p`	18	**16**
`* q;`	`q`	20	

6.2. Manipulation des pointeurs et adresses par les fonctions

L'étude des pointeurs fait l'objet du chapitre 9. On l'évoque ici dans le seul but de permettre l'appel de fonctions standards qui reçoivent en paramètres ou retournent en guise de résultat des adresses.

Exemple. La fonction *scanf()* reçoit en paramètres les adresses des variables où elle doit ranger les valeurs qu'elle lit au clavier.

```
int compteur;           /* initialisation d'un comptage */
.....
printf("\nDonnez la valeur initiale du compteur :  ");
scanf("%d",&compteur);  /* & : adresse de compteur */
```

7. Les tableaux *(première approche)*

7.1. Mécanismes de base

Un **tableau** est une portion d'espace en mémoire, permettant de stocker dans une série de mots contigus plusieurs *variables du même type*. On appelle ***dimension*** d'un tableau le nombre d'éléments qu'il contient.

Déclaration des tableaux

La déclaration d'un tableau s'effectue dans les mêmes formes que celle de n'importe quelle variable, avec l'adaptation suivante : le ***nom*** de variable, commun à tous les éléments, est suffixé entre crochets **`[ ]`** par la ***dimension*** du tableau.[12]

Exemple : `long int total_ventes_du_mois[12];`

La déclaration d'un tableau peut comporter une clause d'***initialisation***. Cette clause énumère, dans une liste entre accolades **`{ }`**, les valeurs initiales de tous les éléments du tableau; chacune de ces valeurs est définie de la manière habituelle, par une constante (ou, plus généralement, par une expression de valeur constante).

```
Exemple :   short int nbre_jours_dans_mois[12]
                 = {31,29,31,30,31,30,31,31,30,31,30,31};
```

Désignation d'un élément de tableau : l'indexation

La désignation d'un élément du tableau se fait au moyen d'une expression formée de deux parties : le ***nom*** unique identifiant tous les éléments, suivi entre crochets **`[ ]`** d'une expression arithmétique appelée ***indice***, dont la valeur, obligatoirement *entière*, indique la position de l'élément dans le tableau.

Si `d` est la dimension du tableau, cette valeur peut s'échelonner de `0` à `d-1`.

nbre_jours_	31	29	31	30	31	30	31	31	30	31	30	31
dans_mois	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*11*

Exemples : *désignation d'un élément :* `nbre_jours_dans_mois[mois-1]`
total des ventes de décembre : `total_ventes_du_mois[11]`

Parcours d'un tableau

Il est fréquent qu'un programme doive parcourir un tableau, c'est-à-dire en examiner ou traiter tous les éléments. Le parcours d'un tableau adopte le schéma d'opération ci-dessous :

```
indice ← première position à examiner (souvent 0 )                    init
TANT QUE indice < dimension (ou dernière position à examiner)          TANT QUE cond
EXECUTER      traiter l'élément désigné par l'indice                   EXECUTER    oper
              indice ← indice + 1                                                  incr
FIN-TANT                                                               FIN-TANT
```

Pour cela, on programmera une boucle ***while*** ou, mieux, une boucle ***for****(init;cond;incr){oper;}* .

Exemple : initialisation des totaux de ventes :

[12] La dimension d'un tableau peut être indiquée par toute expression arithmétique ayant une valeur entière constante. Exemple : `int elt[3*4];`

```
i = 0;
while (i<12)
{
  total_ventes_du_mois[i] = 0;
  ++i;
}
```

```
for (i=0;i<12;++i)
{
   total_ventes_du_mois[i] = 0;
}
```

Exemple : calcul du total des ventes depuis le début de l'année,
le numéro de `mois_en_cours` étant compris entre 1 et 12
rappel : `x+=y` ⇔ `x=x+y`

```
total_ventes_annee = 0;
for (i=0;i<mois_en_cours;++i)
{
   total_ventes_annee += total_ventes_du_mois[i];
}
```

Traitement des tableaux par les fonctions

Le chapitre 9 (traitant des pointeurs) développera les différents aspects du traitement des tableaux par une fonction. On énonce ici deux particularités, dont l'explication sera donnée dans ce chapitre.

1) Lorsqu'on passe un **tableau en paramètre** à une fonction, celle-ci reçoit en réalité l'*adresse* de ce tableau (et non pas une copie du contenu du tableau). En conséquence, si la fonction appelée modifie le contenu du tableau, cette modification altère le contenu du paramètre effectif dans la fonction appelante.

2) Une fonction ne peut pas renvoyer (`return`) un **tableau en résultat**.

Exemple : appel de la fonction de lecture *scanf()* :

```
char texte[80+1];       /* tableau de caractères */
... scanf("%s",texte)         /* garnissage du tableau
            par la lecture d'une chaîne de caractères */
... scanf("%s",&texte)        /* idem - & facultatif */
... scanf("%c",&texte[0])     /* garnissage d'un élément
            par la lecture d'un caractère */
                        /* N.B.  & = "adresse de" */
```

Lorsqu'on programme une fonction opérant sur un tableau dont elle reçoit l'adresse en paramètre, on souhaite habituellement en faire une fonction générale, capable de traiter un tableau de dimension quelconque. Elle doit donc trouver dans ses paramètres une information lui permettant de connaître la dimension effective du tableau; quant à celui-ci, il sera déclaré de dimension indéterminée.

Exemple : prototype d'une fonction ayant pour but d'ordonner (trier) les éléments d'un tableau :

```
void trier (int tableau[], int nbre_elts);
    /* tableau[] : tableau de dimension indéterminée
       nbre_elts : dimension du tableau */
```

7.2. Tableaux à plusieurs dimensions

Il est possible de déclarer un tableau à plusieurs dimensions. On peut ainsi décrire une matrice rectangulaire, un tableau de coordonnées ou une page de papier ... comme étant chacun un tableau de lignes et chaque ligne, un tableau de colonnes. Synthétiquement, il s'agit là de tableaux à deux dimensions. Un tableau peut avoir davantage de dimensions, mais généralement pas plus de trois (pages, lignes, colonnes).

La syntaxe de déclaration d'un tableau à plusieurs dimensions et la syntaxe de désignation d'un élément d'un tel tableau sont analogues : le nom identificateur est suivi d'autant de paires de crochets **[ ]** que le tableau possède de dimensions. Les paires de crochets suivent l'ordre hiérarchique ***[pages] [lignes] [colonnes] ...***

Exemple : *déclaration d'un écran de 25 lignes x 80 colonnes :* `char ecran[25][80];`

déclaration des indices :
```
int ligne;      /* de 0 à 24 */
int colonne;    /* de 0 à 79 */
```

position d'un caractère sur l'écran : `ecran[ligne][colonne]`

position d'une ligne sur l'écran : `ecran[ligne]`

Le parcours d'un tableau utilise autant de boucles ***for*** emboîtées que le tableau possède de dimensions.[13]

Exemple : *constitution de la table de multiplication :*

```
short int mult [10][10];      /* tableau 10x10 */
short int i; short int j;     /* indices */
for (i=0;i<10;++i)
{ /* traitement d'une ligne : */
   for (j=0;j<10;++j)
   { /* traitement d'une colonne : */
      mult [i][j] = (i+1) * (j+1);
   }
}
```

	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
0	1	2	3	4	5	6	7	8	9	10
1	2	4	6	8	10	12	14	16	18	20
2	3	6	9	12	15	18	21	24	27	30
3	4	8	12	16	20	24	28	32	36	40
4	5	10	15	20	25	30	35	40	45	50
5	6	12	18	24	30	36	42	48	54	60
6	7	14	21	28	35	42	49	56	63	70
7	8	16	24	32	40	48	56	64	72	80
8	9	18	27	36	45	54	63	72	81	90
9	10	20	30	40	50	60	70	80	90	100

L'expression d'initialisation d'un tableau à plusieurs dimensions est elle aussi hiérarchisée en un système de listes ***{ }*** emboîtées.

Exemple : *double tableau des mois pour les années bissextiles et non bissextiles :*
```
short nbre_jours_dans_mois[2][12]
     = { {31,29,31,30,31,30,31,31,30,31,30,31},
         {31,28,31,30,31,30,31,31,30,31,30,31} };
```

7.3. Chaînes de caractères

Rappel : une chaîne de caractères est codée dans une suite d'octets contigus (dont chacun contient un caractère), suivis d'un octet contenant, en guise de **terminateur**, un caractère nul (nombre 0 binaire).

`"Bonjour !\n"`

B	o	n	j	o	u	r		!	\n	**\0**

[13] C'est ce nécessaire emboîtement des opérations qui justifie une vision hiérarchisée en lignes et en colonnes ...

Une chaîne de caractères peut donc être stockée dans un ***tableau de caractères*** suffisamment long pour contenir le texte *et son terminateur*.

Exemple : *déclaration :* `char nom_fichier[17] = "A:exemple.txt";`

Un tableau de chaînes de caractères peut être défini par une déclaration de la forme suivante :

```
char nom [nbre_chaînes] [long_max_chaîne] ;
```

Exemple : *déclaration :*

```
char jour[7][9] = { "dimanche",
                    "lundi",
                    "mardi",
                    "mercredi",
                    "jeudi",
                    "vendredi",
                    "samedi"    };
```

	0	1	2	3	4	5	6	7	8
0	d	i	m	a	n	c	h	e	°
1	l	u	n	d	i	°	°	°	°
2	m	a	r	d	i	°	°	°	°
3	m	e	r	c	r	e	d	i	°
4	j	e	u	d	i	°	°	°	°
5	v	e	n	d	r	e	d	i	°
6	s	a	m	e	d	i	°	°	°

désignation de la chaîne "dimanche" : `jour[0]`
désignation de son initiale : `jour[0][0]`

7.4. Initialisation des tableaux. Syntaxe de déclaration simplifiée

Comme pour toute variable, la déclaration d'un tableau peut comporter une liste d'initialisation définissant la valeur initiale de chaque élément.

Un certain nombre de simplifications d'écriture sont possibles.

- **Initialisation partielle**

Si une liste fournit moins de valeurs qu'il n'existe d'éléments à initialiser, les derniers éléments sont mis à la valeur 0. Exemples :

```
char nom_fichier[15] = {'*', '.', '*'};
                  →     * . * \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
```

```
char jour [7][9] = {{'d'},{'l'},{'m'},{'m'},{'j'},{'v'},{'s'}};
```

→ garnit correctement l'initiale de chaque nom de jour de la semaine
le reste de chaque nom est rempli de zéros binaires

- **Initialisation par une chaîne de caractères**

Un tableau de caractères peut être initialisé par une chaîne de caractères plutôt que par une liste de caractères isolés. Exemple :

```
char nom_fichier[15] = "*.*";
                  →     * . * \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0 \0
```

La dimension du tableau peut être telle que le terminateur `\0` n'ait pas de place – ceci est une commodité octroyée pour l'initialisation d'un tableau de caractères qui n'est *pas une chaîne de caractères*. Exemple :

```
char alphabet [26] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ";
```

• **Ajustement automatique des dimensions**

Si la dimension du tableau n'est pas mentionnée, elle est ajustée pour contenir exactement les valeurs initiales fournies. Exemples :

```
short int nbre_jours_dans_mois []
     = {31,29,31,30,31,30,31,31,30,31,30,31};
char nom_fichier[] = "*.*";    /* dimension = [4] ... dont \0 */
```

Cette tolérance n'existe que pour la dimension la plus extérieure, c'est-à-dire pour la première paire de crochets `[]`. Exemple :

```
char jour [][9] = {"dimanche",
                   "lundi",
                   "mardi",
                   "mercredi",
                   "jeudi",
                   "vendredi",
                   "samedi"   };
```

Si la liste d'initialisation d'un tableau à plusieurs dimensions indique les valeurs initiales de *tous* les éléments, les accolades intérieures peuvent être omises. Exemples :

```
short int nbre_jours_dans_mois [2][12]
     = { 31,29,31,30,31,30,31,31,30,31,30,31,
         31,28,31,30,31,30,31,31,30,31,30,31 };
⇔    short int nbre_jours_dans_mois [2][12]
     = {{31,29,31,30,31,30,31,31,30,31,30,31},
        {31,28,31,30,31,30,31,31,30,31,30,31}};

mais char jour [7][9] = {'d','l','m','m','j','v','s'};
⇔    char jour [7][9] = {"dlmmjvs","","","","","",""};
                → indique "dlmmjvs" comme nom de jour[0]
                  et la chaîne vide "" comme nom pour les autres jours
```

Exercices

Types de données

1.

Tester la fonction donnée en exemple pour lister la partie graphique de l'alphabet ANSI. Modifier cette fonction de la manière suivante : déclarer i de type **char** et l'initialiser à *blanc* plutôt qu'à 32.

2.

Créer un programme qui affiche la taille de tous les types scalaires.

3.

En utilisant la fonction *puiss()* du chapitre 1, créer une fonction qui, recevant en paramètre la taille d'un type entier fournie par l'opérateur *sizeof*, affiche les intervalles de valeurs (minimum .. maximum) des variantes signée et non signée de ce type. Ecrire un programme qui teste cette fonction pour chacun des types entiers.

Si, sans compter le signe, on dispose de n bits,

- la plus grande valeur positive représentable est $+(2^n)-1$,
- la plus grande valeur négative représentable est $-(2^n)$.

Constantes

4.

Créer un programme qui affiche la taille des constantes données en exemple au paragraphe 3.

Déclarations

5.

Rédiger et compiler une fonction ne contenant pas d'instructions, mais seulement

- toutes sortes de déclarations de variables et de tableaux,
- des prototypes de fonctions.

Tableaux

6.

Créer un programme qui lit au clavier une date en chiffres sous la forme ***jj mm 19aa*** et la réaffiche avec le nom du mois. Exemple : *4/10/1993* $\rightarrow$ *4 octobre 1993*. Utiliser un tableau des noms de mois. Ecrire ce programme en une seule fonction.

7.

Déclarer un tableau de 12 éléments "total des ventes du mois".
Garnir toutes les positions de ce tableau en appelant la fonction standard *rand()* qui, chaque fois qu'elle est appelée, renvoie un nombre pseudo-aléatoire. [Cette fonction est décrite dans le fichier d'en-tête <*stdlib.h*> de la manière suivante : `int rand(void);`]
Calculer la moyenne des ventes mensuelles. (ATTENTION : quel doit être le type des variables nécessaires à ce calcul ?)
Après cela, afficher sur 12 lignes de l'écran : "ventes du mois *mm* : *nnnnn s*" (*mm* est le numéro du mois, de 1 à 12); *s* est le caractère "+" si *nnnnn* dépasse la moyenne mensuelle ou le caractère "–" si *nnnnn* est inférieur à cette moyenne.
Ecrire ce programme en une seule fonction.

8.

Déclarer et initialiser le tableau illustré ci-dessous.

	0	1	2
0	*Madame*	*Mademoiselle*	*Monsieur*
1	*Mevrouw*	*Juffrouw*	*Meneer*

9.

Un service hospitalier comporte 21 lits. Chaque semaine, il édite un tableau récapitulatif des températures relevées pour l'ensemble des lits tous les jours de la semaine, matin et soir. Une observation manquante est remplacée par 0.
Déclarer ce tableau (quel doit être le type des données ?) et écrire les instructions qui l'initialisent entièrement à zéro.

Chapitre 3. Les entrées et sorties standards

1. Les flots de données standards

Un programme en exécution communique avec son environnement au moyen de **"flots" de données**.

A tout programme, le système d'exploitation associe un **flot standard d'entrée** et un **flot standard de sortie**, assurant la communication avec l'opérateur humain. Dans la bibliothèque C standard, ces flots de données ont pour noms ***stdin*** et ***stdout*** (il existe aussi un flot ***stderr*** des messages d'erreur). Ces flots sont définis dans le fichier d'en-tête standard **<stdio.h>** ("standard input-output") .

Par défaut, sur les équipements actuels,

- le flot standard d'entrée *stdin* est formé des données introduites au clavier;
- le flot standard de sortie *stdout* est formé des données affichées à l'écran.

Vu du programme, un tel flot de données est un texte, c'est-à-dire une suite de ***caractères***, découpé en ***lignes*** (chaque occurrence du caractère `\n` "new line" marque une fin de ligne). Puisqu'il s'agit de communiquer avec un opérateur humain, toutes les ***données*** sont représentées par des caractères visualisables (donc : pas de nombres en représentations binaires `int` ou `float` ...).

2. Fonctions d'accès aux flots standards

Le fichier d'en-tête `<stdio.h>` définit notamment des fonctions d'accès aux flots de données standards. Ces fonctions font l'objet d'une double classification :

• *d'après le flot visé :*

– les fonctions de *lecture* transfèrent des données d'un flot d'entrée vers le programme en mémoire centrale;
– les fonctions d'*écriture* transfèrent vers un flot de sortie des données du programme en mémoire centrale;

• *d'après la quantité d'information transférée à chaque appel :*

– certaines fonctions transfèrent un seul *caractère*;
– certaines fonctions transfèrent le texte d'une *ligne* complète;
– certaines fonctions transfèrent une ou plusieurs *données* décrites par un "format";

	caractère	**ligne**	**données formatées**
lecture *stdin*	`getchar()` *get character*	`gets()` *get string*	`scanf()` *scan with format*
écriture *stdout*	`putchar()` *put character*	`puts()` *put string*	`printf()` *print with format*

Signal de fin de fichier : EOF

Les fonctions de lecture de données d'entrée peuvent retourner à la fonction appelante un signal de fin de fichier. Le même signal est renvoyé en cas d'incident technique. Ce signal est un `int` de *valeur négative* défini dans `<stdio.h>` sous le nom EOF ("end of file").

Pour donner au clavier un signal de fin de fichier, enfoncer simultanément

- dans l'environnement UNIX, les deux touches ***ctrl D*** ;
- dans l'environnement MS-DOS, les deux touches ***ctrl Z*** ;
- dans l'environnement VAX/VMS, les deux touches ***ctrl Z*** .

2.1. Entrée/Sortie d'un caractère – *getchar(), putchar()*

La fonction **`getchar()`** lit le caractère suivant dans le flot *stdin*.

```
int getchar (void);
```

paramètre : néant
retour : le caractère lu – EOF en cas d'incident ou de fin de fichier

La fonction **`putchar()`** écrit un caractère dans le flot *stdout*.

```
int putchar (int caractere);
```

paramètre : le caractère à écrire
retour : le caractère écrit – EOF en cas d'incident

Remarque. Pour les fonctions *getchar()* et *putchar()*, le caractère `\n` de fin de ligne n'a pas d'interprétation particulière; il est traité comme n'importe quel autre caractère.

Ces fonctions placent le caractère transféré dans une variable de type `int`, donc de taille supérieure. Ceci est nécessaire pour pouvoir recevoir un signal EOF qui ne puisse pas être confondu avec un caractère. Illustration :

caractère `\255`	binaire : `|00000000|11111111|`	hexadécimal : `|00|FF|`
signal EOF (`-1`)	binaire : `|11111111|11111111|`	hexadécimal : `|FF|FF|`

Il est évident que les programmes doivent tenir compte de la taille de la donnée en retour.

Exemple :

```
unsigned char caract;
int n;
   /* lecture : */
n = getchar();
if (n!=EOF) caract = n;
   /* écriture : */
n = putchar(caract);
      /* le paramètre est converti automatiquement */
```

2.2. Entrée/Sortie d'une ligne – *gets(), puts()*

La fonction **`gets()`** lit la ligne suivante dans le flot *stdin*.

```
char* gets (char ligne[]);
```

paramètre :	un tableau de caractères qui recevra le texte de la ligne la ligne peut être vide de tout caractère
retour :	l'adresse du tableau[14] – zéro en cas d'incident ou de fin de fichier

La fonction **`puts()`** écrit une ligne dans le flot *stdout*.

```
int puts (char ligne[]);
```

paramètre :	un tableau ou une chaîne de caractères contenant le texte de la ligne la ligne peut être vide de tout caractère
retour :	un nombre positif ou nul – EOF en cas d'incident

En *mémoire centrale*, le texte est mis sous la forme d'une chaîne de caractères et est donc clôturé par *\0*.
Dans le *flot de données*, le texte de la ligne est clôturé par le caractère `\n` de fin de ligne.

Exemple. Le programme suivant copie le flot *stdin* dans le flot *stdout*. Si on en dévie la sortie vers un fichier, il permet de créer à partir du clavier un fichier de texte.

```
#include <stdio.h>
void main(void)          /* copie stdin dans stdout */
{
   char ligne[80+1];           /* 80 caractères + \0 */
   while (gets(&ligne) != 0)         /* & facultatif */
   {  puts(ligne);  }
}
```

2.3. Entrée/Sortie formatée

printf() – "print with format"

La fonction **`printf()`** écrit dans le flot *stdout* une série de données décrites par leurs formats.

```
int printf (char format[], ...);
```

paramètres :	chaîne de caractères décrivant les formats d'affichage suivie de 0 à n valeurs à afficher
retour :	nombre de caractères écrits

La chaîne de contrôle (qui peut être placée dans un tableau de caractères) contient du texte, des commandes de mise en page (ex.: `\n`) et des codes définissant le format sous lequel doivent être affichées les valeurs.

[14] `char*` : pointeur vers un tableau de caractères – cf. chapitre 2.

Une **spécification de format** est composée comme ceci :

%[*présentation*][*largeur*][**.** *précision*][*taille*]*conversion*

Les seules parties obligatoires sont le préfixe % et le code de conversion.

spécification	***code***	***effet***
présentation[a]	**-**	cadrer à gauche plutôt qu'à droite
	+	toujours imprimer le signe algébrique plutôt que seulement si -
	espace	si le premier caractère n'est pas un signe, mettre un espace au début
	0	remplir à gauche par des zéros plutôt que par des espaces
	#	altération du mode de conversion (voir ci-dessous)
largeur	*nombre*[b]	largeur minimum de la zone d'affichage
précision	**.***nombre*[b]	nombre de caractères à afficher (doit être ≤ largeur) selon le mode de conversion : **%s** : nombre maximum de caractères affichés **%f**, **%e**, **%E** : nombre de chiffres après le point[c] **%g**, **%G** : nombre de chiffres significatifs[c] *entière* : nombre minimum de chiffres affichés (complété de zéros à gauche)
taille de la valeur	**h**	("half") *short int* (par défaut, un nombre entier est traité comme *int*)
	l	*long int* (par défaut, un nombre entier est traité comme *int*)
	L	*long double* (par défaut, un nombre réel est traité comme *double*)
conversion	**d** *ou* **i**	("integer") nombre décimal signé
	u	("unsigned") nombre décimal non signé
	o	nombre octal non signé, préfixé par *0* si présentation **#**
	x *ou* **X**	nombre hexadécimal non signé, avec chiffres *a..f* pour **x**, avec chiffres *A..F* pour **X** préfixé par *0x* ou *0X* si présentation **#**
	c	un seul caractère
	s	("string") chaîne de caractères
	f	nombre en virgule flottante : *[-]mmm.ddd*
	e *ou* **E**	nombre réel avec exposant : *[-]m.dddddd*±*xx* ou *[-]m.dddddd*E±*xx*
	g *ou* **G**	format **%f** si l'exposant est proche de 0, sinon format **%e** ou **%E**
	p	pointeur (adresse de mémoire)
	%	caractère *%* (n'est pas un code de conversion)

[a] Les codes de présentation peuvent être donnés dans un ordre quelconque.
[b] Si ce nombre est remplacé par `*`, l'indication est prise dans le paramètre suivant, qui doit être un `int`;
ex.: `printf("| %*d | %*.2f |\n",largeur_qte,qte,largeur_prix,prix);`
[c] Par défaut, la précision est 6. Si la précision est 0, le point décimal est supprimé, sauf si présentation `#`.

scanf() – "scan with format".

La fonction **`scanf()`** lit dans le flot *stdin* une série de données décrites par leurs formats.

```
int scanf (char format[], ...);
```

paramètres :	chaîne de caractères décrivant les formats d'affichage suivie de 0 à n adresses de variables qui recevront les données lues
retour :	nombre de variables garnies – EOF en cas d'incident ou de fin de fichier

La chaîne de contrôle (qui peut être placée dans un tableau de caractères) contient des codes définissant le format sous lequel doivent être lues les valeurs entrées au clavier; elle peut aussi contenir des caractères intercalaires, qui seront traités de la manière indiquée plus loin.

Une **spécification de format** est composée comme ceci :

`%`[*saut*][*largeur*][*taille*]*conversion*

Les seules parties obligatoires sont le préfixe % et le code de conversion.

spécification	***code***	***effet***
saut	`*`	la valeur lue par cette spécification de format est "sautée" (elle n'est affectée à aucune variable de la liste de paramètres)
largeur	*nombre*	nombre maximum de caractères lus au clavier pour cette variable
taille de la variable	`h`	("half") *short int* (par défaut, un nombre entier est traité comme *int*)
	`l`	*long int* (par défaut, un nombre entier est traité comme *int*)
	`l`	*double* (par défaut, un nombre réel est traité comme *float*)
	`L`	*long double* (par défaut, un nombre réel est traité comme *float*)
conversion	`d`	nombre entier décimal signé
	`u`	("unsigned") nombre décimal non signé
	`o`	nombre octal non signé, préfixé ou non par *0*
	`x` *ou* `X`	nombre hexadécimal non signé, préfixé ou non par *0x* ou *0X*
	`i`	("integer") nombre entier sous un des formats suivants : décimal, octal préfixé par *0*, hexadécimal préfixé par *0x* ou *0X*
	`c`	autant de caractères qu'indiqué par largeur (ou, par défaut, un seul) les caractères d'espacement ne sont pas sautés le résultat n'est pas suffixé par le terminateur *\0*
	`s`	("string") chaîne de caractères sans caractères d'espacement[a]
	`e f g`	nombre réel sous le format *[-]mmm[.]ddd[e[±]xx]* ou *[-]mmm[.]ddd[*E*[±]xx]*
	`p`	pointeur (adresse de mémoire)
	`[...]`[b]	la plus longue chaîne exclusivement formée de caractères appartenant à la liste `[...]` [a] `[]...]` permet d'inclure *]* dans la liste
	`[^...]`[b]	la plus longue chaîne formée de caractères n'appartenant pas à la liste `[^...]` [a] `[^]...]` permet d'inclure *]* dans la liste
	`n`	ne lit rien mais indique dans le paramètre correspondant le nombre de caractères lus jusqu'ici (peut être utile quand on doit vérifier la validité du *texte* introduit)
	`%`	caractère *%* intercalaire (n'est pas un code de conversion)

[a] Le résultat est automatiquement suffixé par le terminateur *\0*.

[b] `...` représente une liste de caractères quelconques; exemples : `[+-*/] [^{}89]`

Les données introduites peuvent être librement espacées par des blancs, des tabulateurs ou des sauts de lignes. Toutefois, une lecture sous le format `%c` traite les **caractères d'espacement** comme n'importe quel autre.

Exemple : après avoir demandé une réponse O/N (oui/non),
lire le caractère de réponse et la touche fonction `\n` de fin de réponse

```
printf("Oui/Non ? :"); scanf("%c%*c",&reponse);
```

A chaque **caractère d'espacement** intercalé entre les spécifications de format peut correspondre dans le flot de données une suite de 0 à N caractères d'espacement (blancs, tabulateurs, sauts de lignes), qui ne sont pas rangés dans les variables résultats. Tout autre **caractère intercalé** entre les spécifications de format doit être introduit tel quel au clavier, mais n'est pas non plus rangé dans les variables résultats de la lecture.

Exemple : l'appel suivant demande une date dont les trois parties sont séparées par /

```
scanf("%2hd/%2hd/%2hd",&jj,&mm,&aa);
```

Exemples

Demander à l'opérateur sous quelle forme il veut donner les dates;
sur la base de sa réponse, construire la chaîne décrivant le format.
Demander d'introduire une date.

```
#include <stdio.h>

char format_date[] = "%2hd/%2hd/%2hd";    /* format d'une date */
.....
puts("Vous pouvez introduire les dates sous trois formes :");
puts("   jj/mm/aa   jj-mm-aa   jj mm aa");
printf("Quel caractère de séparation choisissez-vous ? ");
scanf("%c%*c",&format_date[4]);    /* lire sépar. + fin réponse */
format_date[9] = format_date[4];   /* garnir l'autre séparateur */
.....
/* pour lire une date, utiliser une instruction de cette forme :
       scanf(format_date,&jour,&mois,&annee);                  */
.....
```

Pour lire un texte comportant des caractères d'espacement, il faut le clôturer par un caractère terminateur (par exemple, le signe de ponctuation `';'`) et utiliser le format de lecture `%[^;]` :

```
#include <stdio.h>

char auteur[30+1]; /* tableau de réception du nom d'auteur */
char titre[60+1];  /* tableau de réception du titre de l'ouvrage */
.....
     /* lire le nom d'auteur et le titre : */
scanf ("%[^;];%[^;];", &auteur, &titre);
/* exemple :   texte introduit :   André CLARINVAL;Le langage C;
   format : %[^;]  texte lu : "André CLARINVAL"
            ;                 ";" (non rangé dans une variable)
            %[^;]             "Le langage C"
            ;                 ";" (non rangé dans une variable)  */
```

Problèmes

Lire des données en utilisant différentes fonctions de lecture ne provoque normalement pas d'incident technique. Cependant, le mélange de ***scanf()*** et ***gets()*** peut donner des résultats incohérents.

• `scanf("%[^\n]%*c",t)` et `scanf(" %[^\n]%*c",t)` ont un comportement proche de celui de `gets(t)`. Cependant, la lecture d'une ligne vide au moyen du premier format n'effacera pas le contenu antérieur du tableau `t`. L'espace initial dans le second format prescrit de passer outre des caractères d'espacement (y compris les éventuels caractères de fin de ligne) présents au début du texte lu; ce format ne permet pas de lire une chaîne vide ou uniquement composée de caractères d'espacement – *scanf()* va "insister" jusqu'à ce qu'on lui donne un caractère non blanc !

• Si l'on veut lire une ligne par *gets()* alors que la précédente l'a été par *scanf()*, on peut exécuter deux fois l'appel *gets()* – le premier lit la fin de la ligne précédente, `\n` compris.

3. Déviation des flots standards

3.1. Déviation par le langage de commande

Les flots standards ***stdin*** et ***stdout*** peuvent être "**déviés**" vers des fichiers. Le moyen d'effectuer cette redéfinition est fourni par le langage de commande propre à chaque ordinateur.

Dans les systèmes UNIX et MS-DOS, la commande d'exécution du programme (préalablement constitué par le relieur) doit être libellée de la manière suivante :

nom_programme **<** *nom_fichier_entrée* **>** *nom_fichier_sortie*

Ne doivent être indiqués que les opérateurs de déviation < > et noms de fichiers nécessaires.

Exemples : exécution du programme *listalph.exe* avec production d'une liste *alphabet.lis* :

```
listalph >alphabet.lis
```

exécution du programme *calcul.exe* prenant ses données dans le fichier *test.dat* :

```
calcul <test.dat
```

idem, avec mise des résultats dans un fichier *calcul.tst* :

```
calcul <test.dat >calcul.tst
```

Dans l'environnement VAX/VMS, la commande RUN d'exécution du programme doit être précédée de la redéfinition des noms de flots standards :

```
$ DEFINE SYS$INPUT nom_fichier_entrée
$ DEFINE SYS$OUTPUT nom_fichier_sortie
$ RUN nom_programme
```

Dans les systèmes UNIX et MS-DOS, le flux *stdout* d'un premier programme peut servir d'entrée *stdin* à un second programme; le flux *stdout* du deuxième programme peut servir d'entrée *stdin* à un troisième programme, et ainsi de suite. Ce mécanisme s'appelle un "**tube**" ("pipe", en anglais). Pour cela, on écrit :

program1 **|** *program2* **|** *program3* ...

3.2. Déviation programmée – fonction *freopen()*

Supposons un programme qui se déroule en trois phases :

1) *introduction de données par dialogue au terminal (écran + clavier);*
2) *calculs sur ces données;*
3) *création d'une liste imprimée des résultats.*

Supposons que la liste des résultats doive être conservée; on l'écrit donc dans un fichier qui pourra être réimprimé à tout moment.

La fonction *printf()* est utilisée à la phase 1 pour produire des affichages à l'écran et à la phase 3 pour écrire dans le fichier des résultats. Entre ces deux phases, le flot de sortie standard `stdout` doit être dévié vers le fichier; la fonction ***freopen()*** *"file re-open"* est utilisée dans ce but :

• *déviation des sorties standards :*

```
freopen ("désignation du fichier", "w", stdout);     /*  "w" = "write"  */
```

• *déviation des entrées standards :*

```
freopen ("désignation du fichier", "r", stdin);      /*  "r" = "read"  */
```

Exemple :

```
#include <stdio.h>
int main (void)
{
 /* phase 1 : dialogue */
   .....
 /* phase 2 : calculs */
   .....
 /* phase 3 : résultats */
   freopen ("resultat.lis", "w", stdout);
   .....
}
```

Exercices

1.

Réaliser toutes sortes de dialogues au terminal :

- demander d'introduire des données,
- lire ces données,
- les réafficher à titre de vérification.

(Exemple : demander les éléments d'une étiquette d'adresse puis mettre en forme cette étiquette.)

Tester notamment toutes sortes de formats pour *printf()* et *scanf()*.

2.

Dévier les résultats (par exemple, l'étiquette d'adresse) dans un fichier à imprimer.

Chapitre 4. Les expressions

1. Définitions

1.1 Eléments d'une expression

Les expressions (arithmétiques – ex.: `prix_net*taux_tva/100` , logiques – ex.: `n>1` , etc.) constituent une forme syntaxique fondamentale des langages de programmation et, particulièrement, de C.

Le problème résolu par une expression est le suivant : *construire une nouvelle valeur sur la base de valeurs préexistantes*. Ces valeurs préexistantes, qu'on appelle ***opérandes***, sont combinées au moyen d'***opérateurs***, par exemple des opérateurs arithmétiques.

Particularité du langage C : la grande diversité des **opérateurs** disponibles, dont voici quelques-uns des plus usuels :

- opérateurs *arithmétiques* : `+` `-` `*` (multiplication) `/`
- opérateurs *de comparaison* : `==` (égal) `!=` (différent)
  `<` `<=` (inférieur ou égal) `>=` (supérieur ou égal) `>`
- opérateurs *logiques* : `&&` (et) `||` (ou)

Un **opérande** peut prendre une des formes suivantes :

- *constante* ex.: `2.5` `'Z'` `32000`
- identificateur de *variable* `i` `t[i]` `prix_un` `qte`
- appel de *fonction* `puiss(n,2)`
- sous-*expression*, placée ou non entre parenthèses **( )** `-5` `(a * b / 100)`

Par *généralisation*, on peut dire que :

- un opérande (une valeur) constitue à lui seul une expression;
- tout opérande est donc une sous-expression de l'expression qui le contient;
- une fonction est un opérateur créé par le programmeur, dont les opérandes sont les paramètres (un *appel* de fonction est une sous-expression).

Déterminer la valeur (du résultat) d'une expression se dit **évaluer** une expression.

1.2. Catégories d'expressions

Il existe en C des **expressions d'adressage**, dont le résultat est une adresse de mémoire; nous les étudierons plus tard. Dans ce chapitre, nous nous intéresserons aux **expressions numériques** dont le résultat est une valeur numérique, entière ou réelle ... .

> On a défini au chapitre 2 une première expression d'adressage : l'***indexation*** `t[i]`, `t[i][j]` ... d'un élément de tableau. En lui-même, un indice, placé entre `[ ]`, est une *expression numérique* dont le résultat est une valeur d'un *type entier*.

Une expression (numérique ou d'adressage) dont tous les opérandes sont des constantes produit un résultat qui est forcément constant lui-même ... et qui peut être connu par le compilateur lorsqu'il analyse le texte du programme. Le compilateur peut faire certains usages particuliers d'une **expression constante** – on en a déjà vu un exemple : la détermination de la valeur initiale d'une variable.

> **Remarque.** Par raccourci, les qualificatifs s'appliquant au résultat d'une expression sont employés pour qualifier l'expression elle-même; on parlera donc directement du "type d'une expression" et on dira semblablement : "expression entière", "expression constante", etc.

1.3. Usage des expressions numériques

Sous-expression en opérande

L'évaluation d'une expression numérique produit une **valeur** de résultat que, en principe, le programme récupère pour l'*affecter*

• à une variable par une opération d'*affectation*	ex.: `a_payer = prix_un * quantite`
• à un paramètre lors d'un *appel de fonction*	`puiss(n, longueur - 1)`
• comme résultat au *retour d'une fonction*	`return ((a * b) / pgcd(a,b))`
• comme opérande d'une *autre opération*	`prix * (taux / 100)`

Dans tous ces cas, l'expression (en italique, dans les exemples ci-dessus) joue le rôle d'**opérande** ou sous-expression d'une expression englobante.

Une expression (évaluée comme *vraie* ou *fausse*) peut également servir d'opérande à une instruction de **test** `if` ou `while`.

Exemples :
```
while (n > 0) ...
if (code_sexe == 'M') ...
```

Instruction

Evaluer une expression consiste à exécuter certaines **actions**. C'est une particularité du langage C qu'on puisse parfois évaluer une expression simplement pour en exécuter les actions, sans transmettre la valeur qui en résulte à une expression englobante. On utilise alors l'expression en guise d'**instruction**, ordre donné à l'ordinateur d'exécuter certaines actions.

Exemple : un appel de fonction est une expression ...
- appel d'une fonction en tant qu'opérande : `n2 = puiss(n,2);`
- appel d'une fonction en tant qu'instruction : `printf("Bonjour \n");`
  la valeur retournée par la fonction (nombre de caractères affichés) n'est pas récupérée

2. Ordre d'évaluation des expressions

Quelle est la valeur de l'expression `12+8/2` ? Est-ce *10* ou *16* ? ...

Pour toute expression comportant plusieurs opérateurs, il est nécessaire de dicter l'ordre dans lequel les opérations doivent être effectuées ou évaluées.

A cette fin, les représentations mathématiques habituelles exploitent les possibilités d'un espace à deux dimensions :

$$\frac{12+8}{2} \qquad 12+\frac{8}{2}$$

L'écriture unidirectionnelle d'un programme nécessite d'autres mécanismes.

2.1. Emploi des parenthèses

Le programmeur peut enfermer une sous-expression entre parenthèses **( )**. Cette sous-expression doit constituer par elle-même une expression correctement formée.

Toute sous-expression entre parenthèses est évaluée avant que son résultat serve d'opérande à l'expression contenante.

$\frac{12+8}{2}$ ⇔ `((12 + 8) / 2)`

évaluation :

`(12 + 8)` → `20`

`(20 / 2)` → `10`

$12+\frac{8}{2}$ ⇔ `(12 + ( 8 / 2))`

évaluation :

`( 8 / 2)` → `4`

`(12 + 4)` → `16`

2.2. Priorité et associativité des opérateurs

En l'absence de parenthèses, le compilateur détermine l'ordre d'évaluation des opérations en appliquant les règles de *priorité* et d'*associativité* définies par le langage.

Priorité

Les opérateurs sont classés suivant des niveaux de **priorité**. L'évaluation d'une expression (ou d'une sous-expression entre parenthèses) commence par les opérations ayant la plus haute priorité et finit par celles qui ont la priorité la plus basse.

Exemple : parce que l'opérateur / de division est davantage prioritaire que l'opérateur + d'addition,

```
12+8/2 ⇔ (12+(8/2)) → 16
```

Associativité

Lorsque se succèdent plusieurs opérateurs de même priorité, l'évaluation procède le plus souvent de gauche à droite; dans quelques cas, elle procède de droite à gauche. Cette direction est ce qu'on appelle l'**associativité** des opérateurs.

Exemple : + (addition) et − (soustraction), de même priorité, s'associent de gauche à droite

```
posit_debut + longueur - 1
⇔ (posit_debut + longueur) - 1
```

On verra que, pour plusieurs opérateurs, la direction de l'associativité n'est pas sans importance.

Tableau général des opérateurs

Priorité	*Groupes d'opérateurs*	*Opérations*[a]	*	*Notation*	*Associativité*
max					
15	suffixes	**appel de fonction**	*	`x(y,...)`	→
		indexation	*	*`x[y]`*	
		sélection de membre	*	*`x.id`*	
		sélection de membre	*	*`x->id`*	
		post-incrémentation	*	`x++`	
		post-décrémentation	*	`x--`	
14	préfixes	**pré-incrémentation**	*	`++x`	←
		pré-décrémentation	*	`--x`	
		adresse de	*	*`&x`*	
		indirection	*	*`*x`*	
		plus		`+x`	
		moins		`-x`	
		complément sur bits		`~x`	
		négation logique	*	`!x`	
		taille de	*	`sizeof x`	
		conversion au type	*	`(TYPE)x`	
13	θ multiplicatifs	multiplication		`x*y`	→
		division		`x/y`	
		modulo (reste)		`x%y`	
12	θ additifs	addition	*	`x+y`	→
		soustraction	*	`x-y`	
11	θ de décalage	décalage à gauche		`x<<y`	→
		décalage à droite		`x>>y`	
10	relationnels	supérieur	*	`x>y`	→
		supérieur ou égal	*	`x>=y`	
		inférieur	*	`x<y`	
		inférieur ou égal	*	`x<=y`	
09	d'égalité	égal	*	`x==y`	→
		différent	*	`x!=y`	
08	θ booléens sur bits	ET (intersection)		`x&y`	→
07		OU exclusif		`x^y`	
06		OU inclusif (union)		`x|y`	
05	logiques	ET (produit)	*	`x&&y`	→
04		OU (somme)	*	`x||y`	
03	conditionnel	conditionnel	*	`x?y:z`	←
02	affectation	**affectation absolue**	*	`x=y`	←
		affectation relative	*	`xθ=y` [b]	
01	séquentiel	séquentiel	*	`x,y`	→
min					

* certains opérandes peuvent (ou doivent) être des pointeurs

[a] *en italique*, les opérateurs d'*adressage* en **gras**, les opérateurs effectuant une **affectation**

[b] θ : tout opérateur pris dans les groupes marqués de ce signe

3. Le type des opérandes. Conversions de types

Lors de l'évaluation d'une expression, il se peut que certains opérandes soient convertis dans un autre type que leur type d'origine.

Ces conversions sont principalement justifiées par la taille de la représentation des valeurs.

3.1. L'opérateur *sizeof*

L'opérateur **sizeof** renvoie, sous la forme d'un entier, la taille en nombre d'octets de la valeur de son opérande. Cet opérande est

- soit une ***expression***

  pour la lisibilité, il est *recommandé* de placer l'expression entre parenthèses

  exemples :
  ```
  sizeof (qte)    sizeof(234e-12)    sizeof(heure*3600)
  sizeof (prix * (1 + taux / 100))
  ```

- soit une spécification de ***type*** – la taille indiquée est commune à toutes les valeurs de ce type

  une spécification de type prend la forme d'une *déclaration de variable sans identificateur*
  il est *obligatoire* de placer la spécification du type entre parenthèses

  exemples :
  ```
  sizeof(float)    sizeof(unsigned long int)
  sizeof(int[40])   /* tableau de 40 données int */
  ```

Remarque. **`sizeof()`** est une fonction interne *du compilateur*. Son résultat, connu dès le stade de la compilation et non modifiable par l'exécution du programme, est donc une constante.

3.2. Conversions forcées

Dans le cas d'une opération pour laquelle, d'une manière ou d'une autre, le programmeur fixe lui-même le type du résultat *attendu*, le résultat *"calculé"* est converti dans le type demandé.

Une conversion forcée peut provoquer une perte d'information. Par exemple, le passage d'un type réel à un type entier entraîne la perte de la partie fractionnaire du nombre d'origine.

Opérations d'affectation

Une conversion forcée se produit dans les différents cas d'**affectation** à une variable (de type déterminé) de la valeur produite par l'évaluation d'une expression.

Exemples : – les deux exemples ci-dessous opèrent la troncature entière d'un nombre réel :

- *instruction d'affectation :*
  ```
  { float n; int i;
    i = n; }
  ```
- *affectation du résultat d'une fonction :*
  ```
  int tronque (float n)
  { return n; }
  ```

– conversion d'un nombre entier en réel lors de l'appel d'une fonction
(affectation des paramètres effectifs aux paramètres formels) :

- *déclaration de la fonction :* `float puiss(float n,int p);`
- *appel de la fonction :* `... puiss(2,3) ...  /* 2 -> 2.0F */`

Opérateur de coercition (TYPE)

En dehors des cas d'affectation, on peut forcer la conversion d'un opérande d'un certain type dans un autre type, en faisant précéder cet opérande du nom du type destinataire entre parenthèses : `(TYPE)x`. Cette construction est connue sous le nom anglais de "cast operator" (opérateur de "moulage") et, en français, sous le nom d'opérateur de ***coercition***.

Exemple : *opérateur de conversion au type* int *:* `... (int)n ...`
le but de la fonction *tronque* ci-dessus serait plus apparent si on écrivait :

```
int tronque (float n)
{ return (int)n; }
```

Les opérateurs de conversion s'associent de droite à gauche. On peut donc écrire :

```
float fraction = 3.5;
... (float)(int)fraction ...
```

ce qui s'évalue : `(int)3.5` → `3`
`(float)3` → `3.0`

3.3. Conversions implicites

Dans le cas d'une opération pour laquelle le type du résultat n'est pas indiqué par le programmeur (ex.: *a* + *b*), le compilateur assure des conversions implicites. Elles sont de deux sortes.

Promotion entière

Cette conversion, *qui a toujours lieu*, reflète le privilège que le langage C accorde au type **int**.

Tout opérande d'un type entier de taille plus courte que le type *int* est converti dans le type **`signed int`**.

```
signed char    → signed int
unsigned char  → signed int
signed short   → signed int
unsigned short → signed int    si sizeof(short) < sizeof(int)
```

mais :

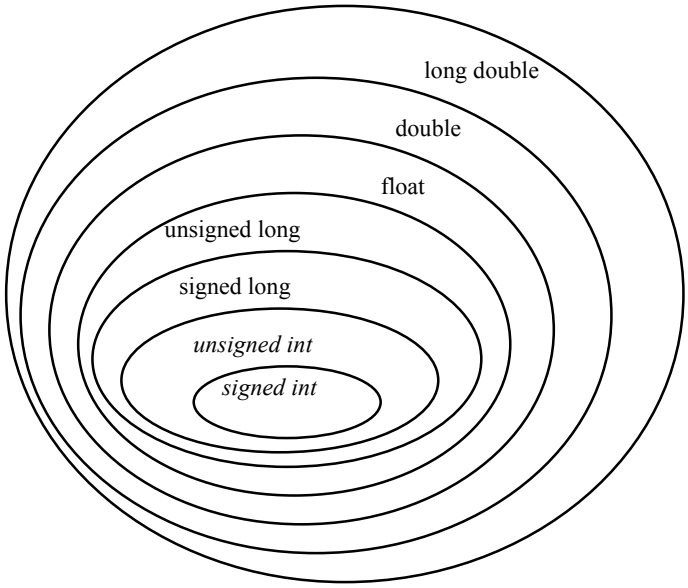
```
unsigned short ⇔ unsigned int  si sizeof(short) = sizeof(int)
```

Conversion arithmétique

Lorsqu'un opérateur, tel qu'un opérateur arithmétique, combine deux opérandes, il se produit une conversion dont le but est double : représenter les deux opérandes dans *le même type,*[15] *en évitant les pertes d'information*. Le principe est simple : ***le type commun est celui des deux dont le domaine de valeurs est le plus étendu***.

[15] En cela, le langage C imite les langages assembleurs.

Domaines de valeurs :



Remarque. Aucune conversion ne doit être opérée d'un type ***int*** à un type ***long***, si ces deux types ont la même taille.

ATTENTION ! Une conversion peut se produire d'un type entier ***signé*** au type **non signé de même taille**. La norme du langage ne définit pas quel est le résultat de cette opération pour le cas d'une valeur de départ négative (elle ne dit donc pas qu'on obtient la valeur absolue !).

Rappel. Quoique leur valeur ne soit jamais négative, les **constantes entières** sont d'un type signé, sauf si elles comportent le suffixe **U** ou si la valeur est si grande qu'elle ne peut s'exprimer que dans le type `unsigned long int`.

S'il y a lieu, la *promotion entière* est d'abord appliquée aux deux opérandes; la *conversion arithmétique* s'effectue ensuite.

4. Les calculs, au sens large

4.1. Opérations arithmétiques

Opérateurs

Le langage C fournit les opérateurs arithmétiques de base, dont voici la liste :

Priorité	*Groupes d'opérateurs*	*Opérations*	*Notation*	*Associativité*
14	préfixes	valeur opposé	`+x` `-x`	←

13	multiplicatifs	multiplication
		division
		modulo (reste de x/y)
			`x*y`
			`x/y`
			`x%y`
				→

12	additifs	addition soustraction	`x+y` `x-y`	→

Exemples

```
- n
i+1
(position_debut + longueur - 1)    /* -> position de fin */
(n * -1)    ⇔    n * (-1)
3.0 / 7.0
millesime % 4       /* ... est-ce une année bissextile ? */
'z' + ('A' - 'a')       /* -> 'Z' dans l'alphabet ASCII */
prix*(1-pct_remise[categ_client]/100)
      ⇔    prix*(1-((pct_remise[categ_client])/100))
1 + rand() % 42     /* - la fonction standard 'rand()'
                         donne un entier pseudo-aléatoire
                      - l'expression ramène ce nombre
                         dans l'intervalle [1..42]       */
```

Opérandes et résultat

Les opérandes sont d'un type entier ou réel, sauf pour l'opération *x*`%`*y* dont les deux opérandes doivent avoir un type entier.

Si les deux opérandes de la division *x*`/`*y* sont d'un type entier, le résultat est un entier; sinon le quotient est un nombre réel. Exemple : `3.0/4.0` → `0.75` mais `3/4` → `0`.

Les opérandes subissent la promotion entière et, dans le cas des opérations binaires, la conversion arithmétique. Le type du résultat est le même que celui des opérandes après conversion.

REMARQUE IMPORTANTE. Le résultat d'une opération, particulièrement une multiplication ou une division, doit parfois se représenter dans un type différent de celui des opérandes. Il faut alors forcer un des opérandes au type voulu pour le résultat.

Exemples

Soit les déclarations : `int prix_de_base = 333; int pct_remise = 15;`

```
(prix_de_base * pct_remise) / 100         → 49
prix_de_base * (pct_remise / 100)         → 0
```

Toutes les évaluations seraient correctes si l'un des opérandes était un nombre réel :

```
(prix_de_base * (float)pct_remise) / 100  → 49.95
prix_de_base * ((float)pct_remise / 100)  → 49.95
```

```
int heure = 24;
... heure * 3600 ...      /* -> secondes dans une journée */
```

Les deux opérandes étant du type *int*, le résultat est lui-même du type *int*.
– Si la taille d'un *int* est 32 bits, le résultat sera mathématiquement correct.
– Si la taille d'un *int* est 16 bits, le résultat sera erroné.
Solution : donner à l'un des opérandes le type *long int* :

```
... (long)heure * 3600 ...  ou  ... heure * 3600L ...
```

Remarques à propos de la division

`x/y` donne le quotient et `x%y` donne le reste de la division de `x` par `y`.

Le résultat de ces opérations est indéfini si le diviseur vaut 0.

Si les deux opérandes de la division `x/y` sont d'un type entier, le résultat est un entier; sinon le quotient est un nombre réel.

Dans la division entière, le quotient est le quotient par défaut; il n'est jamais arrondi à l'unité supérieure.

Si l'un au moins des opérandes de `x%y` est négatif, le signe + ou − du résultat n'est pas déterminé par la norme du langage.

Exemples :

```
8 / 5      → 1        8 / 2      → +4
8.0 / 5    → 1.6      8 / -2     → -4
8 % 5      → 3        -8 / 2     → -4
                      -8 / -2    → +4
```

4.2. Comparaisons

Opérateurs

Priorité	*Groupes d'opérateurs*	*Opérations*	*Notation*	*Associativité*

10	relationnels	supérieur
		supérieur ou égal
		inférieur
		inférieur ou égal
			`x>y`
			`x>=y`
			`x<y`
			`x<=y`
				→

09	d'égalité	égal différent	`x==y` `x!=y`	→

Exemples

```
a == b                                  ATTENTION ! L'opérateur du test d'égalité
c[i] == 0                                           est formé de deux signes =
i != j+1     ⇔     i != (j+1)
c[i] == c[i-1]     /* deux fois le même caract. consécutivement */
i <= 2 * j - 1
(j>0)
pgcd(a,b) != 1
i <= posit_debut+longueur-1    ⇔    i<=(posit_debut+longueur-1)
```

Opérandes et résultat

Les opérandes sont d'un type entier ou réel. Il subissent la promotion entière et la conversion arithmétique.

Le résultat d'une comparaison est de type ***int*** et vaut
- **0** si la comparaison est **fausse**,
- **1** si la comparaison est **vraie**.

... Puisque le résultat d'une comparaison est un nombre entier ... on peut comparer les résultats de deux comparaisons. (On pourrait même faire sur eux des opérations arithmétiques, mais cela n'aurait aucun sens.)

Exemple : l'expression suivante constitue un *OU exclusif* entre deux comparaisons :
l'expression complète est vraie si, des deux comparaisons,
l'une – n'importe laquelle – est vraie et l'autre fausse : `(a==b) != (a==c)`

Associativité

En C, les opérateurs de comparaison sont associatifs; on peut donc écrire des expressions telles que `a>b>0` ou `a!=b==c` ... Ces manières de faire inhabituelles sont à proscrire, car leur résultat est rarement celui qu'on croit ! Illustrations :

expressions	*évaluation* (→)	*interprétation*
`8 > 4 > 2`	`8 > 4` → `1` `1 > 2` → `0`	faux
`2 < 4 < 8`	`2 < 4` → `1` `1 < 8` → `1`	vrai
`6 == (3*2 == 2+4)`	`3*2 == 2+4` → `1` `6 == 1` → `0`	faux

4.3. Opérations logiques

Opérateurs

Priorité	*Groupes d'opérateurs*	*Opérations*	*Notation*	*Associativité*
14	préfixes	négation (complément logique)	`!x`	←
05	logiques	ET (produit logique)	`x&&y`	→
04		OU (somme logique)	`x||y`	

Comme les opérations arithmétiques combinent les valeurs numériques, les opérations logiques combinent les **valeurs logiques *vrai* et *faux*.**

Les tableaux ci-dessous indiquent la valeur des expressions suivant la valeur logique des opérandes. (Les tableaux de droite donnent l'équivalent numérique utilisé par l'***algèbre de Boole***, sur la base des équivalences suivantes : *faux = 0, vrai = 1*.)

• **négation** (complément logique) **`!x`** : → *vrai* si l'opérande est faux; *faux* dans l'autre cas

`x`	`! x`
faux	vrai
vrai	faux

`x`	`1 - x`
0	1
1	0

• **ET** (produit logique) **`x&&y`** : → *vrai* si les opérandes sont tous les deux vrais; *faux* dans les autres cas
si le premier opérande est faux, le second n'est pas évalué

`x`	`y`	`x && y`
faux	faux	*faux*
faux	vrai	*faux*
vrai	faux	faux
vrai	vrai	vrai

`x`	`y`	`min(x,y)`
0	0	0
0	1	0
1	0	0
1	1	1

• **OU** (somme logique) **`x||y`** : → *vrai* si un opérande au moins est vrai; *faux* dans les autres cas
si le premier opérande est vrai, le second n'est pas évalué

`x`	`y`	`x || y`
faux	faux	faux
faux	vrai	vrai
vrai	faux	*vrai*

`x`	`y`	`max(x,y)`
0	0	0
0	1	1
1	0	1

vrai	vrai	*vrai*

1	1	1

Exemples

tests de validité :

```
(mois > 0) && (mois <= 12)     ⇔     (mois>0 && mois<=12)
                               ⇔     !(mois<=0 || mois>12)
sexe=='M' || sexe=='F'
c>='A' && c<='Z' || c>='a' && c<='z'      /* c est une lettre */
      ⇔     (((c>='A') && (c<='Z')) || ((c>='a') && (c<='z')))
```

Opérandes et résultat

Les opérandes sont d'un type entier ou réel. Il subissent la promotion entière et, dans le cas des opérations binaires, la conversion arithmétique.

Un **opérande** est considéré comme
- **faux** si sa valeur est ***égale à 0***,
- **vrai** si sa valeur est ***différente de 0***.[16]

Le **résultat** est de type ***int*** et vaut
- **0** si la combinaison est **fausse**,
- **1** si la combinaison est **vraie**.

Les opérations réellement effectuées sont donc les suivantes :

`x`	`! x`
`== 0`	`1`
`!= 0`	`0`

`x`	`y`	`x && y`	`x || y`
`== 0`	`== 0`	`0`	`0`
`== 0`	`!= 0`	`0`	`1`
`!= 0`	`== 0`	`0`	`1`
`!= 0`	`!= 0`	`1`	`1`

Ambiguïté des valeurs logiques

> **C'est une particularité du langage C de ne pas avoir à proprement parler de valeurs logiques "vrai"/"faux", mais d'effectuer une *interprétation logique de valeurs numériques*.**

Il est donc permis d'écrire `if(!a)` au lieu de `if(a==0)` !! Et il n'est pas rare, dans les programmes C, de lire la tournure `while (c[i])` ... ce qui signifie `while (c[i]!=0)` !!

Remarques

NE PAS OUBLIER de doubler les symboles **`&&`** et **`||`** (`&` et `|` simples sont d'autres opérateurs).

La règle de l'**évaluation partielle**, en vertu de laquelle le second opérande n'est quelquefois pas évalué, est techniquement nécessaire pour l'opérateur `&&`, afin de rendre possible la recherche dans un tableau ou un fichier d'un élément non nécessairement présent : *TANT que le numéro d'élément est inférieur au nombre d'éléments présents (test toujours possible) ET que l'élément désigné n'est pas celui qu'on cherche (test impossible lorsque le précédent donne le résultat "faux"), incrémenter le numéro d'élément.*

[16] Le langage C s'écarte ici de l'algèbre de Boole, pour laquelle "vrai" est figuré par le seul nombre 1.

Exemple : chercher la première occurrence de 0 dans un tableau :

```
for (i=0;(i<dimens_tableau && n[i]!=0);++i){}
/* après cette instruction, i "pointe"
   - soit sur la posit. immédiatement après le tableau
   - soit sur le 1er. 0 à l'intérieur du tableau */
/* remarquer le traitement vide {}
   à chaque répétition de la boucle */

/* variante :  && n[i]  <=>  && n[i]!=0
                  ;     <=>     {}       ... */
for (i=0;(i<dimens_tableau && n[i]);++i);
```

4.4. Manipulation des bits

Introduction

Il n'est pas négligeable, le nombre d'informations qui peuvent être représentées dans un espace de mémoire plus petit qu'un octet, c'est-à-dire avec moins de 8 bits. ***Un espace de* n *bits suffit pour représenter* 2^n *informations mutuellement exclusives.***

Exemples

Un seul bit, par ses deux valeurs 0 et 1, peut représenter tout couple d'attributs mutuellement exclusifs, tels que *vrai/faux, présent/absent, chaud/froid, en ligne/déconnecté, global/local, déjà/pas encore traité ...*

Le système d'exploitation VMS de l'ordinateur VAX associe à tout fichier quatre *masques de protection* décrivant les droits d'accès accordés à quatre catégories d'utilisateurs : le responsable du *système*, l'utilisateur *propriétaire* du fichier, le *groupe* d'utilisateurs auquel appartient le propriétaire, le "*monde*" des autres utilisateurs. Chaque masque est formé de quatre bits indiquant si le droit correspondant est ou non octroyé à l'utilisateur – ces droits d'accès sont les droits de *lire* ("Read"), *écrire* ("Write"), *exécuter* ("Execute"), *effacer* ("Delete") le fichier; en abrégé : "*rwed*".[17]

16 bits	15			0
masque	system	owner	group	world
droits	r w e d	r w e d	r w e d	r w e d

Le langage de programmation COBOL autorise différents modes de *représentation des nombres* : représentation *binaire* (comme en C), sous la forme d'une *chaîne de caractères* dont chacun est un chiffre décimal, ou sous une forme *décimale condensée* ("packed decimal") dans laquelle un chiffre décimal est représenté dans un demi-octet (ce qui est possible, puisque $9_{10} = 1001_2 < 2^4$).

Pendant qu'il analyse le texte d'un programme, un compilateur constitue une table des déclarations qu'il rencontre. Un compilateur C pourrait coder sur 4 bits seulement le type d'une variable :

bits :
0-1 : mode de représentation : 00 = entier non signé, 01 = entier signé, 11 = réel (signé)
2-3 : taille = 2^n octets : n = 0, 1, 2, 3

[17] Le système UNIX utilise une technique analogue : trois masques (le responsable du système possède toujours tous les droits) de trois bits "*rwx*" (le droit d'effacer fait partie du droit d'écrire).

Opérateurs

A l'instar des langages assembleurs dont il veut rester proche, le langage C fournit des opérateurs agissant sur des groupes de bits à l'intérieur d'un mot de mémoire.

Priorité	*Groupes d'opérateurs*	*Opérations*	*Notation*	*Associativité*
14	préfixes	complément sur bits	`~x`	←
11	de décalage	décalage à gauche décalage à droite	`x<<y` `x>>y`	→
08	booléens sur bits	ET (intersection)	`x&y`	→
07		OU exclusif	`x^y`	
06		OU inclusif (union)	`x|y`	

Opérations de décalage

Les opérations de décalage décalent les bits contenus dans le premier opérande d'un nombre *p* de positions, indiqué par la valeur du second opérande.

L'opérateur << effectue un décalage **vers la gauche**. Les *p* bits de droite sont mis à 0.

```
Illustrations :   1 << 1 → 2        00000001  →  00000010
                  1 << 2 → 4        00000001  →  00000100
                  3 << 1 → 6        00000011  →  00000110
                  3 << 2 → 12       00000011  →  00001100
```

L'opérateur >> effectue un décalage **vers la droite**. Selon les options propres à chaque type d'ordinateur, les *p* bits de gauche sont mis à 0 (décalage "logique") ou prennent la même valeur que le bit de signe (décalage "arithmétique").

Opérations booléennes

Les opérations booléennes sur bits appliquent séparément sur chaque bit de leurs opérandes les fonctions de l'algèbre de Boole. Les opérations à deux opérandes combinent entre eux les bits occupant la même position dans les deux opérandes.

- **complément** `~x` : inverse chaque bit de l'opérande;
- **ET** (intersection) `x&y` : laisse inchangées les positions qui sont à 1 dans les deux opérandes à la fois, force les autres à 0;
- **OU inclusif** (union) `x|y` : laisse inchangées les positions qui sont à 0 dans les deux opérandes à la fois, force les autres à 1;
- **OU exclusif** `x^y` : force à 0 les positions qui ont la même valeur dans les deux opérandes, force à 1 les positions qui ont une valeur différente dans les deux opérandes.

Le tableau ci-dessous indique quel est le résultat pour chaque couple de bits.

`x`	`~ x`
	1 - x
0	1
1	0

`x`	`y`	`x & y`	`x | y`	`x ^ y`
		min(x,y)	*max(x,y)*	
0	0	0	0	0
0	1	0	1	1
1	0	0	1	1
1	1	1	1	0

Habituellement, le premier opérande est la variable manipulée (testée ou mise à jour), tandis que le second opérande est une constante, donnée sous forme hexadécimale, qui joue le rôle d'un masque appliqué sur la variable.

Exemples

```
/* N.B. les bits sont numérotés de droite à gauche à partir de 0 */
/* pour la compréhension, dans les exemples ci-dessous,
   la valeur de chaque expression est affectée à une variable */

masque = 1 << p;      /* masque identifiant le bit en position p */
if (var & masque) ... /* teste le bit p --> vrai si 1 */[18]
var = var | masque;   /* met le bit p à 1 quel que soit son état */
var = var & ~masque;  /* met le bit p à 0 quel que soit son état */

if (var & 1)          /* var est-il impair ? */

/* extraire la valeur codée dans un groupe de bits contigus :
   cadrer à droite puis masquer les positions excédentaires */
v = mot >> 5 & 3;     /* valeur [0..3] des bits 5-6 d'un mot */
v = mot >> 4 & 7;     /* valeur [0..7] des bits 4-6 d'un mot */

char octet[17];
char chiffre[33];
..... /* extraire les deux chiffres décimaux codés dans un octet
         d'un nombre décimal condensé */
chiffre[p*2+1] = (octet[p] & 0xff) + '0';
chiffre[p*2] = (octet[p] >> 4 & 0xff) + '0';
      /* l'opération +'0' transforme le chiffre
         en caractère ASCII : 0 + '0' --> '0'
                              1 + '0' --> '1' ... */

/* permuter 2 éléments d'un tableau
   sans employer de zone de manoeuvre */
a[i] = 5; a[j] = 3;     /* 00000101  00000011 */
a[i] = a[i] ^ a[j];     /* 00000110           */
a[j] = a[j] ^ a[i];     /*           00000101 */
a[i] = a[i] ^ a[j];     /* 00000011           */
```

Opérandes et résultat

Les opérandes doivent être d'un type entier. Ils subissent la promotion entière, et dans le cas des opérations booléennes, la conversion arithmétique. Le type du résultat est le même que celui du premier opérande après conversion.

Dans une opération de décalage << ou >>, la valeur du second opérande doit être positive ou nulle et ne peut être supérieure au nombre de bits du premier opérande.

[18] Pour que le test d'un bit puisse s'exprimer brièvement par la formule `if(var&masque)`, plutôt que par `if((var&masque)!=0)`, il faut que tout nombre comportant un bit quelconque à 1 – c'est-à-dire, en pratique, tout *nombre différent de 0* – soit *considéré comme vrai*. En ceci se trouve la justification de cette option caractéristique du langage C.

5. Les opérations d'affectation

5.1. Affectation absolue[19]

Opérateur

• Format :	*var* **=** *val*

• Opération : la valeur résultat de l'expression de droite (*val*)
est rangée dans la variable désignée par l'expression de gauche (*var*).

Exemples

```
salaire_net = (salaire_brut - cotisations) - impot
taxe = prix * taux / 100
n2 = puiss(n,2)
b[i] = 1 + rand() % 42
```

Opérandes et résultat

L'expression de gauche désigne une variable.[20]

Avant d'être rangée dans la variable réceptrice, la valeur fournie par l'expression de droite est convertie dans le type de cette variable.

Exemples :
```
unsigned int u;
signed int i;
signed int j;
float n;
.....
i = u;      /* conversion unsigned --> signed */
n = i;      /* conversion int --> float */
j = n;      /* conversion float --> int */
```

Ambiguïté des opérations d'affectation

L'affectation – modification du contenu d'une variable – est un mécanisme fondamental de tous les langages de programmation.

Originalité du langage C : une opération d'affectation est également une *expression* et peut donc constituer une sous-expression d'une expression englobante; elle "retourne" alors une valeur à l'expression englobante.

L'effet d'une opération d'affectation *var=val* est donc double :

1) la valeur de *val* est convertie dans le type de *var* puis rangée dans *var*;
2) la nouvelle valeur de *var* (après l'éventuelle conversion de type) est "retournée" au contexte.

On peut schématiser ce double effet de la manière suivante : *contexte* ← *var* ← *val*

---

[19] Le qualificatif *absolue* est employé par opposition à la qualification d'*affectation relative*, définie au paragraphe suivant.
[20] La mise à jour de la variable ne doit pas être interdite par le qualificatif *const* – cf. chapitre 7.

... Il est donc permis d'écrire des expressions du genre de celles-ci, véritables défis pour l'esprit !

```
c[j=i+1]    masque = 1 << (p = longueur % 8)
```

Cependant, effectuer dans une même expression ***l'affectation d'une valeur à une variable et le test de cette valeur*** peut simplifier l'écriture du programme.

Exemple. Le programme ci-dessous copie le flot *stdin* dans *stdout*, en convertissant les majuscules en minuscules.

```
#include <stdio.h>      /* déf. getchar(), putchar(), EOF */
#include <ctype.h>      /* déf. tolower() --> lettre minuscule */

void main (void)
{
   int c;          /* caractère converti en int par getchar() */
   while ( (c=getchar()) != EOF ) putchar(tolower(c));
}
```

Associativité

Des sous-expressions d'affectation s'associent de droite à gauche.

Exemple : `c = b = a = 0` $c \leftarrow b \leftarrow a \leftarrow 0$

Erreur fréquente : confusion des opérateurs d'affectation et d'égalité

Il arrive facilement qu'on oublie de doubler le signe **==** d'un test d'égalité. On écrit ainsi une expression d'affectation, qui "retourne" à son contexte une valeur numérique ... à laquelle il sera donné une interprétation logique !

Exemples : `if (a = 0)` ... est toujours faux ... et la valeur de `a` est modifiée
`if (a = 2)` ... est toujours vrai ... et la valeur de `a` est modifiée
`if (a = b)` ... ne dépend pas de la valeur de `a` mais seulement de la valeur de `b` (nul ?)

5.2. Affectation relative[21]

Opérateurs

• Format : *var* **θ=** *val*

θ est un des opérateurs binaires suivants :

- arithmétiques : `* / %` (multiplication, division, modulo)
  `+ -` (addition, soustraction)
- de manipulation des bits : `<< >>` (décalage)
  `& ^ |` (masques booléens)

[21] Les opérations d'affectation relative du langage C sont les équivalents de certaines instructions de base présentes dans tous les langages assembleurs.

• Opération : *var θ= val* n'est rien d'autre qu'une écriture abrégée pour *var = var θ val*.

Comme on le voit, le résultat n'est pas indépendant de la valeur précédente de la variable désignée par l'expression de gauche; pour cette raison, on parle d'affectation *relative*.

Cas d'emploi

L'intérêt des expressions d'affectation relative réside dans le fait que l'expression de gauche doit être évaluée une seule fois plutôt que deux.

Exemples

```
n[abs(i)][abs(j)] /= 2;   /* l'affectation relative permet
                             d'appeler 2 fois au lieu de 4
                             la fonction abs() - valeur absolue -
                             pour calculer les indices */

prix += taxe;      ⇔    prix = prix + taxe;
longueur *= sizeof(float);

var |= (1 << p);    /* met le bit p à 1 quel que soit son état */
var &= ~(1 << p);   /* met le bit p à 0 quel que soit son état */

/* permuter 2 éléments d'un tableau
   sans employer de zone de manoeuvre */
a[i] = 5; a[j] = 3;     /* 00000101  00000011 */
a[i] ^= a[j];           /* 00000110           */
a[j] ^= a[i];           /*           00000101 */
a[i] ^= a[j];           /* 00000011           */
```

Associativité

!! Par suite de l'associativité des opérations d'affectation, le dernier exemple ci-dessus pourrait s'écrire :

```
a[i] ^= a[j] ^= a[i] ^= a[j];
```

5.3. Incrémentation, décrémentation[22]

Opérateurs

Priorité	*Groupes d'opérateurs*	*Opérations*	*Notation*
15	suffixes	post-incrémentation post-décrémentation	x**++** x**--**
14	préfixes	pré-incrémentation pré-décrémentation	**++**x **--**x

[22] Les opérateurs d'incrémentation/décrémentation sont les équivalents de certains opérateurs du langage assembleur de l'ordinateur (PDP) sur lequel a été créé le langage C.

L'opérande désigne une variable d'un type autorisé pour l'addition ou la soustraction.[23]

– L'opération affecte à cette variable sa propre valeur augmentée (++) ou diminuée (--) de 1.
– Quant à la valeur "retournée" au contexte, elle dépend de la position de l'opérateur :

- les opérations de pré-incrémentation `++x` et pré-décrémentation `--x`
  transmettent la valeur de `x` postérieure à sa mise à jour (la mise à jour est *pré*alable);
- les opérations de post-incrémentation `x++` et post-décrémentation `x--`
  transmettent la valeur de `x` antérieure à sa mise à jour (la mise à jour est *post*érieure).

> Lorsque le contexte n'utilise pas la valeur qui lui est renvoyée, il est préférable de programmer des pré-opérations `++x` ou `--x`; le programme peut, dans ce cas, faire l'économie de la sauvegarde ou de la récupération de la valeur antérieure de `x`.

Exemples

```
for (i=0;i<dimens;++i) ...   /* schéma de parcours d'un tableau */
                  /* ++i préférable à i++ */

   /* écrire des lignes numérotées */
int compteur=0;       /* valeur initiale = 0 */
char texte[40+1];     /* tableau pour chaîne de 40 caractères */
.....
printf ("%3d  %s\n", ++compteur, texte);
      /* le compteur est incrémenté =avant= d'être écrit */

   /* créer l'alphabet majuscule inversé */
char c[26+1];        /* tableau pour 26 lettres + terminateur \0 */
int posit;           /* indice */
char lettre;
...
lettre = 'Z'; posit = 0;
while (posit<26) c[posit++] = lettre--;
      /* l'indice et la valeur évoluent
         =après= le placement de chaque lettre */
c[posit] = '\0';  /* posit vaut 26, càd. la longueur parcourue */
```

Cas d'emploi

Habituellement, les opérations d'incrémentation/décrémentation manipulent un compteur, un indice ... Elles peuvent cependant s'utiliser pour les nombres réels.

RECOMMANDATION. Eviter de mélanger dans une même partie de programme les *pré*-opérations et *post*-opérations. Ce genre de mélange est difficilement maîtrisable par l'esprit.

Priorité

La priorité des opérateurs d'incrémentation et décrémentation est telle qu'il n'est jamais nécessaire de placer l'opérande entre parenthèses.

[23] La mise à jour de la variable ne doit pas être interdite par le qualificatif *const* – cf. chapitre 7.

Associativité

Les opérations d'incrémentation/décrémentation ne sont pas associatives. La nature de leur opérande (désignateur d'une variable) rend la chose impossible : une première incrémentation ou décrémentation ne peut servir d'opérande à une seconde.

6. Les appels de fonctions

Un ***appel*** de fonction est une *expression*, dont les opérandes sont constitués des paramètres de l'appel. (Il est dès lors légitime de considérer la ***fonction*** elle-même comme un opérateur créé par le programmeur.)

Un appel de fonction peut avoir pour paramètre effectif un appel de fonction – les appels de fonctions peuvent s'emboîter. Exemple : `puiss(2,abs(e));`

Comme les opérations d'affectation, l'appel d'une fonction modifie le contenu de certaines variables. Ces affectations se produisent aux deux étapes suivantes : la passation de la valeur des *paramètres* effectifs à la fonction appelée, le renvoi de la valeur *résultat* à la fonction appelante.

Les valeurs ainsi échangées entre les fonctions appelante et appelée subissent des conversions de type.

6.1. Passation des paramètres à la fonction appelée

L'appel d'une fonction se fait sous la forme suivante :

identificateur-de-fonction **(** *liste d'expressions* **)**

Les parenthèses sont obligatoires, mais la liste d'expressions peut être vide.

Chaque expression de la liste entre parenthèses est un **paramètre effectif** de l'appel. La *valeur* de ce paramètre est copiée dans le **paramètre formel** correspondant déclaré dans la fonction. (Les paramètres se correspondent par leur position dans la liste.)

```
appel :                 puiss (      r,      e)   paramètres effectifs
                                     ↓       ↓
déclaration :    float puiss (float n, int p)     paramètres formels
```

Dans le texte du programme, l'appel d'une fonction est en principe précédé de la définition (ou du prototype) de cette fonction.[24] Cette définition préalable de la fonction appelée (ou son absence) dicte les conversions de type subies par les paramètres.

- La valeur d'un paramètre effectif est forcée au type du paramètre formel correspondant si le prototype de la fonction appelée mentionne le type de ce paramètre formel.

ex.: *prototype :* `float puiss (float r, int e);`
appel : `... puiss (2,3)` → *`puiss (2.0F, 3)`*

[24] Plus précisément, la déclaration d'une fonction (ou son rappel) doit être *visible* à tout endroit du programme où cette fonction est appelée. Sur le concept de *visibilité*, cf. chapitre 7.

• Pour les autres paramètres, la conversion de type est implicite :

– un paramètre effectif d'un type entier subit la *promotion entière*;
– un paramètre effectif de type `float` est converti dans le type `double`;
– un paramètre effectif d'un autre type ne subit aucune conversion;
... quel que soit le type attendu par la fonction appelée !!

6.2. Renvoi du résultat à la fonction appelante

Le résultat d'une fonction est renvoyé à la fonction appelante par l'instruction suivante :

```
return expression ;
```

• Si la déclaration de la fonction indique le type du résultat,
la valeur de l'expression est convertie dans ce type.

```
ex.:    long int tronque(float n)
        { return n; }     /* troncature entière d'un nombre réel
                             - résultat de taille 'long int' */
```

• Sinon, par défaut, le résultat est converti dans le type `int`.

7. Les expressions algorithmiques

7.1. Expression conditionnelle (opérateur `?:`)

Opérateur

• Format : *expr1* **?** *expr2* **:** *expr3*

• Opération : si *expr1* ≠ 0 (vrai), le résultat est la valeur d'*expr2* (*expr3* n'est pas évalué)
si *expr1* = 0 (faux), le résultat est la valeur d'*expr3* (*expr2* n'est pas évalué)

Remarque. *expr1* sert de condition. Pour la lisibilité du programme, il est recommandé de la placer entre parenthèses – en effet, dans tous les autres contextes, une condition doit toujours être placée entre parenthèses.

Exemples

```
int abs (int i)
/* fonction retournant la valeur absolue d'un entier */
{ return ( (i<0) ? -i : i ); }
```

```
char majusc (char c)    /* alphabet ASCII :
      fonction convertissant une minuscule en majuscule
      et laissant tel quel tout autre caractère */
{ return ( (c>='a' && c<='z') ? c+('A'-'a') : c ); }
/* propriété de l'alphabet ASCII :
   les lettres majuscules, aussi bien que les minuscules,
   occupent des positions contiguës;
   par conséquent, ('A' - 'a') représente l'écart invariant
   entre une majuscule et la minuscule correspondante */
```

Opérandes et résultat

Tous les opérandes subissent la promotion entière. Les deux derniers opérandes subissent la conversion arithmétique et le résultat est du même type que ces opérandes après conversion.

Associativité

Les deux derniers opérandes (*expr2* : *expr3*) d'une expression conditionnelle peuvent être eux-mêmes des expressions conditionnelles. Il est possible, de cette manière, d'emboîter des expressions conditionnelles.

> *Exemple :* la version récursive de fonction *pgcd()* de calcul du plus grand commun diviseur de deux nombres peut être récrite de la manière suivante :

```
int pgcd (int a, int b)
{
   return (a < b) ? pgcd(a, b - a)
                  : (b < a) ? pgcd(b, a - b)
                            : a                  /* (a == b) */
          ;
}
```

7.2. Expression séquentielle (opérateur ,)

Opérateur

• Format :	*expr1* , *expr2*

• Opération : *expr1* est évaluée en premier lieu,
mais sa valeur n'intervient pas dans la détermination du résultat;
expr2 est évaluée en second lieu, sa valeur devient le résultat de l'opération.

Cas d'emploi

expr1 est exécutée comme une action préalable à l'évaluation d'*expr2*. Pour être utile, cette action préalable doit comporter l'affectation d'une valeur à une variable ou effectuer un échange d'information avec l'utilisateur du programme.

Syntaxiquement, une expression séquentielle peut être utilisée là où est autorisée une expression simple, notamment en guise de condition.

Exemple

```
/* forcer l'utilisateur à donner une réponse autre que 0 : */
while ( scanf("%d",&n), !(n!=0) );   /* répéter la lecture */
```

Associativité

L'opération est associative. Le nombre de sous-expressions successives est donc illimité. Dans tous les cas, le résultat est simplement la valeur de la dernière sous-expression.

Exemple

Le programme ci-dessous demande à son utilisateur d'introduire (→), un à un, une suite de nombres, terminée par 0; après quoi, il affiche le total.

```
#include <stdio.h>

void main (void)
{
   int n;
   long int total = 0;
   while (printf("-> "), scanf("%d",&n), n!=0) total += n;
   printf("total = %ld",total);
}
```

Priorité

L'opérateur , est celui qui a la priorité la plus basse. Il n'est donc jamais nécessaire de placer un de ses opérandes entre parenthèses. Mais, lorsqu'une expression séquentielle sert d'opérande, elle doit elle-même toujours être mise entre parenthèses.

8. Effets de bord

Les règles d'associativité définissent l'ordre d'évaluation des opérations (c'est-à-dire des opérateurs) dans une expression. Sauf dans le cas des expressions algorithmiques, l'ordre d'**évaluation des opérandes** d'une opération n'est pas défini.

Ceci peut entraîner des effets indésirables, appelés *effets de bord*, dans le cas où la valeur d'un opérande dépend de la valeur de l'autre. Ce qui peut arriver *lorsqu'un opérande est une sous-expression d'affectation ou d'appel de fonction qui modifie la valeur de l'autre opérande*.

Exemple : `i = 4;` *évaluation (=)* → *évaluation (=)* ←
`c[++i] = i*2;` ⇔ `c[5]=10` *ou* `c[9]=8`

RECOMMANDATION. On ne doit jamais désirer un effet indésirable... cela piégerait tous les lecteurs !

9. *Supplément.* Quelques fonctions standards

9.1. Fonctions mathématiques

Liste des fonctions (fichier <math.h>*)*

Le fichier d'en-tête ***math.h*** définit des fonctions mathématiques standards. En voici un certain nombre.

Les paramètres et le résultat de ces fonctions sont tous de type `double`.
Pour les fonctions trigonométriques, les angles sont exprimés en radians.

`sin(x)`	sinus de x
`cos(x)`	cosinus de x
`tan(x)`	tangente de x
`asin(x)`	arc sinus de x, dans l'intervalle $[-\pi/2, \pi/2]$, $x \in [-1, 1]$
`acos(x)`	arc cosinus de x, dans l'intervalle $[0, \pi]$, $x \in [-1, 1]$
`atan(x)`	arc tangente de x, dans l'intervalle $[-\pi/2, \pi/2]$
`sinh(x)`	sinus hyperbolique de x
`cosh(x)`	cosinus hyperbolique de x
`tanh(x)`	tangeante hyperbolique de x
`exp(x)`	fonction exponentielle e^x
`log(x)`	logarithme népérien : $\ln(x)$, $x > 0$
`log10(x)`	logarithme à base 10 : $\log_{10}(x)$, $x > 0$
`pow(x,y)`	x^y erreur si $x = 0$ et $y \leq 0$ ou si $x < 0$ et y n'est pas un entier
`sqrt(x)`	$\sqrt{x}$, $x \geq 0$
`fabs(x)`	valeur absolue $\|x\|$, exprimée dans le format *double*
`ceil(x)`	le plus petit entier $\geq x$, exprimé dans le format *double*
`floor(x)`	le plus grand entier $\leq x$, exprimé dans le format *double*

Gestion des erreurs (fichier <errno.h>*)*

Pour vérifier le résultat de l'appel d'une fonction de la bibliothèque ***math.h***, le programme doit tester la variable `errno` (*"error number"*).

Cette variable est définie dans le fichier d'en-tête ***errno.h***, lequel définit également les valeurs codées de ce signal d'erreur. Ces valeurs sont désignées par les noms suivants :

`EDOM`	erreur de domaine : la valeur d'un paramètre est invalide
`ERANGE`	erreur d'intervalle : la valeur du résultat ne peut être représentée dans un *double*

Exemple

```
#include <math.h>       /* fonctions mathématiques */
#include <errno.h>      /* gestion des erreurs */
#include <stdio.h>      /* entrée-sortie des données */

void main (void)
{
   float x; float y;
   double puiss;
   printf ( "TEST de la fonction standard pow()\n"
            "donnez la racine et l'exposant >>>> " );
   scanf ("%f %f", &x, &y);
   puiss = pow(x,y);
   (errno == EDOM || ERANGE) ? printf ("calcul impossible")
                             : printf ("résultat = %g", puiss);
}
```

Liste des fonctions (fichier <stdlib.h>*)*

Le fichier d'en-tête ***stdlib.h*** – *"standard library"* fournit toutes sortes de fonctions utilitaires. Parmi celles-ci, les suivantes peuvent être classées comme fonctions "mathématiques".

- ***Valeur absolue***

`int abs (int x);`	valeur absolue de *x*, sous le format *int*
`long labs (long x);`	valeur absolue de *x*, sous le format *long*

- ***Génération de nombres aléatoires***

`int rand (void);`	génération d'un nombre pseudo-aléatoire
`void srand (int seed);`	initialisation d'une suite de nombres

Chaque appel de la fonction ***rand()*** rend un nombre `int` ≥ 0 pseudo-aléatoire ("random"). L'algorithme interne de cette fonction développe une suite de nombres à partir d'une "graine" ("seed") qui, par défaut, vaut 1. Pour une même "graine", la suite de nombres sera toujours la même.

Il est donc recommandé d'initialiser la suite de nombres en fournissant, par la fonction ***srand()***, une valeur de départ elle-même changeante. Manière simple : prendre la valeur par laquelle la fonction *time()* représente la date et l'heure de l'instant présent :

```
#include <stdlib.h>     /* déf. de rand(), srand() */
#include <time.h>       /* déf. de time() */
   .....
   srand(time(0));   /* donne une valeur de départ pour rand() */
   .....
```

9.2. Gestion des caractères (fichier <*ctype.h*>)

Le fichier d'en-tête standard ***ctype.h*** – *"character type"* définit des fonctions "`is...`" de test d'un caractère et deux fonctions "`to...`" de conversion d'un caractère.

Les fonctions de test renvoient un `int` à 0 (*faux*) ou différent de 0 (*vrai*).

`isprint(c)`	caractère imprimable, y compris `' '`
`isgraph(c)`	caractère imprimable, sauf `' '`
`isdigit(c)`	chiffre décimal
`islower(c)`	lettre minuscule anglaise (non accentuée)
`isupper(c)`	lettre majuscule
`isalpha(c)`	⇔ `islower(c) || isupper(c)`
`isalnum(c)`	⇔ `isdigit(c) || isalpha(c)`
`ispunct(c)`	⇔ `isgraph(c) && !isalnum(c)`
`isspace(c)`	caractère d'espacement : `'\f' '\n' '\r' '\v' '\t' ' '`
`iscntrl(c)`	caractère de contrôle (dans l'alphabet ASCII, positions 0 à 31, 127)
`isxdigit(c)`	chiffre hexadécimal

`tolower(c)`	si `c` est une majuscule, le convertit en minuscule (non accentuée)
`toupper(c)`	si `c` est une minuscule (non accentuée), le convertit en majuscule

Exercices

Opérations arithmétiques

1.

Ecrire une fonction qui, recevant en paramètre l'indication d'une durée en nombre de secondes, l'affiche à l'écran sous la forme *hh:mm:ss* (heures : minutes : secondes) – la taille des variables doit être suffisante pour exprimer l'heure *23:59:59* .

2.

Déclarer un tableau de 12 éléments "total des ventes du mois" – choisir un type de données capable de représenter des montants de l'ordre du million.
Garnir ce tableau par un dialogue à l'écran.
Après cela, afficher sur 12 lignes de l'écran : "ventes du mois *mm* : *nnnnn = pp %*" (*mm* est le numéro du mois, de 1 à 12); *pp* est le pourcentage, arrondi, du total annuel des ventes.

Ecrire ce programme en une seule fonction, en utilisant pour seules variables, en plus d'un indice *mois*, les totaux mensuels et annuel.

3.

Un numéro de compte bancaire est composé de 3 parties : *bbb-nnnnnnn-cc*.
Les 3 premiers chiffres *bbb* identifient la banque;
les 7 chiffres suivants *nnnnnnn* forment le numéro de compte auprès de cet organisme;
les 2 derniers chiffres *cc* composent un nombre de contrôle,
égal au reste de la division par 97 du nombre obtenu en collant les 10 chiffres *bbbnnnnnnn*.

Ecrire une fonction qui, recevant en paramètres les deux nombres *bbb* et *nnnnnnn*, calcule et renvoie *cc*. Exécuter tout le traitement au moyen de la seule instruction `{return` ***expression ;*** `}` .

Pour tester cette fonction, créer un programme qui demande au terminal les deux premières parties d'un numéro de compte et affiche en retour le nombre de contrôle. Paramétrer les fonctions de dialogue *printf()* et *scanf()* de manière telle que le numéro de compte apparaisse à l'écran sous cette forme : *bbb-nnnnnnn-cc* .

Comparaisons

4.

Vérification d'un numéro de compte bancaire. Ecrire une fonction qui, recevant en paramètres les trois nombres *bbb, nnnnnnn, cc* renvoie le résultat logique (0 ou 1) du test "la valeur de *cc* est-elle exacte ? ". Exécuter tout le traitement au moyen de la seule instruction `{return` ***expression ;*** `}` .

Ecrire un programme de test pour cette fonction.

Opérations logiques

5.

Compléter la fonction ci-dessous par la formule exprimant la condition suivante : une année est bissextile si le millésime est un multiple de 4, sans être le millésime d'un siècle; toutefois, les millésimes multiples de 400 sont des années bissextiles.

```
int fevrier(int annee)
{ if (expression) return (29); else return (28);   }
```

Appels de fonctions

6.

Récrire l'exercice 4 de manière telle que l'expression appelle la fonction créée à l'exercice 3.

7.

Rédiger la fonction *tan()* définie dans la bibliothèque standard `<math.h>`, en utilisant les autres fonctions de cette bibliothèque et sachant que la tangente d'un arc est le quotient de son sinus par son cosinus. Exécuter tout le traitement au moyen de la seule instruction `{return` ***expression;*** `}` .

8.

Récrire l'exercice 2 en appelant pour les calculs deux fonctions *pourcent()* et *arrondi()*. Programmer ces fonctions.

Expressions conditionnelles

9.

Récrire l'exercice 5 en exécutant tout le traitement au moyen de la seule instruction `{return` ***expression;*** `}`.

10.

Ecrire une fonction *intmax()* qui, recevant en paramètres deux nombres de type `int`, renvoie le plus grand des deux et une fonction *intmin()*, qui renvoie le plus petit des deux nombres. Dans les deux cas, exécuter tout le traitement au moyen de la seule instruction `{return` ***expression;*** `}` .

Affectations, Expressions séquentielles

11.

Récrire l'exercice 3 (calcul du nombre de contrôle d'un numéro de compte bancaire) pour tenir compte de la précision suivante : *si la valeur calculée pour le nombre* cc *est 0, prendre 97* pour résultat. Exécuter tout le traitement au moyen de la seule instruction `{return` ***expression;*** `}` , en effectuant une seule fois les calculs nécessaires. Deux versions sont possibles : avec ou sans expression séquentielle.

Opérations sur bits. Affectations. Incrémentation/Décrémentation

12.

Demander d'introduire au clavier une chaîne de caractères quelconque. L'afficher en inversant l'ordre des caractères (exemple : *Saint-Laurent* → *tneruaL-tniaS*).

Ecrire une fonction distincte pour effectuer l'inversion du texte. Cette fonction reçoit en paramètre le tableau contenant le texte lu.

Permuter les caractères dans le texte d'origine, sans recourir à des zones de manoeuvre.
Faire évoluer les indices par des opérations d'*incrémentation/décrémentation*.

ATTENTION !
Limiter le traitement aux caractères précédant le terminateur *\0*.
Quand la permutation doit-elle cesser ? Cette condition vaut-elle pour les deux cas où le nombre de caractères est pair ou impair ?

Chapitre 5. Le contrôle de séquence

1. Introduction

1.1. Concepts

Revenons à la définition d'un **algorithme** : ***méthode de composition d'opérations*** pour arriver à la solution certaine de tout problème appartenant à une classe bien définie.

Expressions

Cette définition présume qu'il existe des **opérations de base**. Le langage C propose une large gamme d'***opérateurs***, auxquels il convient d'ajouter les ***fonctions*** standards et les fonctions créées par le programmeur.

On a vu au chapitre 4 qu'une **expression** est une méthode de composition de ces opérations de base. La grande diversité des opérateurs disponibles rend ce mécanisme très puissant.

Exemple : l'algorithme de la fonction *ppcm()*, donnée au chapitre 1, tient en une seule expression.

```
int ppcm (int a, int b)
{ return ((a * b) / pgcd (a, b)); }
```

Ce mécanisme est néanmoins limité par le fait qu'une expression, aussi complexe soit-elle, est capable de créer *une seule valeur*.

Il est donc nécessaire de disposer d'autres mécanismes de composition des opérations, des mécanismes que l'on dit de **contrôle de séquence**.

Instructions

Les opérations combinées par un mécanisme de contrôle de séquence s'appellent des **instructions**. Une instruction est une *formule syntaxique prescrivant l'exécution de certaines actions*.

Une ***expression*** est une première forme d'instruction. Pour pouvoir combiner sans restriction plusieurs expressions, le résultat de chacune doit être mémorisé ... dans une *variable*. Un nombre quelconque d'instructions peuvent manipuler les mêmes variables : lire leur contenu ou le modifier.

Seules, les expressions modifiant l'état de certaines variables ou effectuant un échange d'information entre le programme et son environnement sont utiles en tant qu'instructions. Tels sont les différentes formes d'***expressions d'affectation*** et certains ***appels de fonctions***.

Exemples :
```
n2 = puiss(n,2);
printf("Bonjour \n");
```

Constructions

Tout programme peut être obtenu par l'**emboîtement** de trois types de **constructions** : la *séquence*, l'*itération*, la *sélection*.

• La **séquence** est une suite d'instructions exécutées l'une après l'autre. La forme d'une séquence est la suivante :

pseudo-code	*langage C*
DEBUT	{
opérations	*instructions*
FIN	}

• L'**itération** est une construction provoquant la répétition de l'exécution d'une séquence d'instructions. La forme de base de l'itération est la suivante :

pseudo-code	*langage C*
TANT QUE *condition*	while (*condition*)
EXECUTER *séquence*	{ *instructions* }
FIN-TANT	

L'exécution de la séquence d'instructions se répète tant que la condition est vraie (la condition est testée avant chaque exécution de la séquence). Pour que la répétition finisse par s'arrêter, la séquence répétée doit normalement contenir des instructions capables de rendre fausse la condition.[25]

Les langages de programmation proposent tous quelques variantes de constructions itératives. Outre ***while***, C propose les constructions ***for*** et ***do while***.

• La **sélection** est une construction qui sélectionne et exécute une séquence d'instructions parmi plusieurs possibles. La forme de base de la sélection est la suivante :

pseudo-code	*langage C*
SI *condition*	if (*condition*)
ALORS *séquence*	{ *instructions* }
[SINON *séquence*]	[else { *instructions* }]
FIN-SI	

La première séquence d'instructions est exécutée dans le cas où la condition est vraie, la seconde séquence est exécutée dans le cas contraire. Une de ces séquences peut être vide; si la deuxième séquence est vide, le texte entre crochets peut être omis.

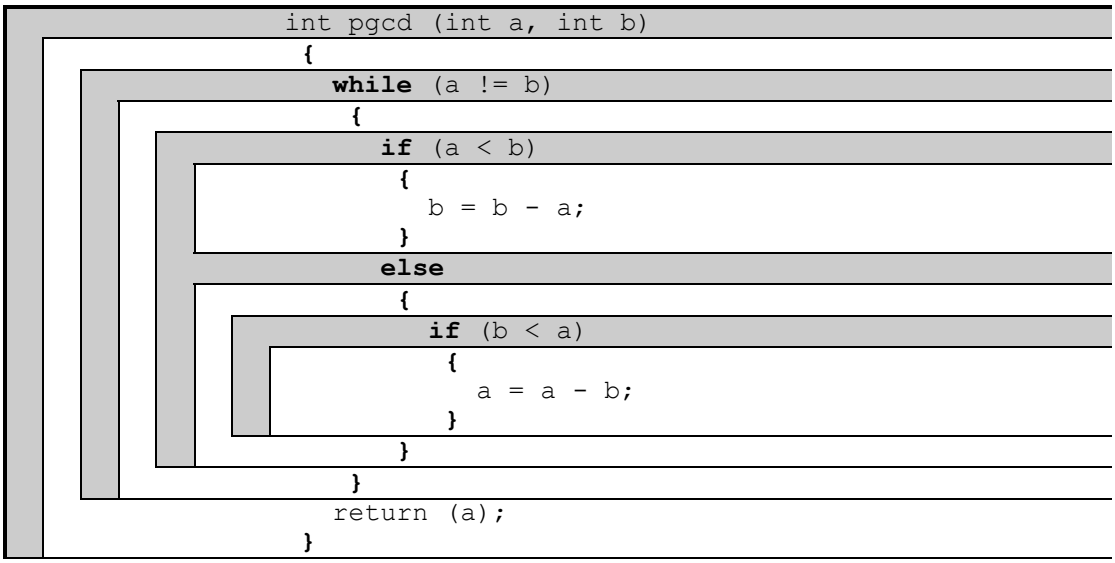
Les langages de programmation proposent une variante, connue sous le nom anglais générique de "case structure", où le nombre de séquences sélectionnables (le nombre de "cas") n'est pas limité à 2; en revanche, le type de condition est souvent restreint. En C, il s'agit de la construction ***switch*** ("aiguillage").

Pour que ces constructions puissent **s'emboîter**, *il faut et il suffit que toute construction soit elle-même considérée comme étant une instruction*. (Les schémas ci-dessus faisaient déjà cette interprétation pour la séquence.)

---

[25] Exception possible : une intervention extérieure au programme. En C, ce peut être la modification d'une variable *volatile* testée dans la condition (cf. chapitre 7).

Exemple : la version non récursive de la fonction *pgcd()*, donnée au chapitre 1, est formée d'un emboîtement de constructions des trois types.



1.2. Conventions générales du langage C

Terminateur d'instruction ;

Toute *instruction de base* (instruction-expression ou instruction de saut – voir plus loin) doit être terminée par un point-virgule.

Par elle-même, une *construction* n'est pas clôturée par un point-virgule.

Bloc { }

La définition, le **corps d'une fonction**, prend la forme d'un bloc { }.

En C, comme dans tous les langages de la lignée d'ALGOL, la construction séquentielle, connue sous le nom de **bloc**, présente la particularité de pouvoir contenir la déclaration de ***variables locales***, seulement accessibles aux instructions de ce bloc. Les déclarations doivent précéder les instructions :

```
{
  déclarations
  instructions
}
```

Le bloc constituant le corps d'une fonction contient les déclarations visibles pour toutes les instructions de la fonction; il est relativement rare que les blocs emboîtés à l'intérieur d'une construction en contiennent.

Si un bloc (autre que le bloc formant le corps d'une fonction) contient, en tout et pour tout, une seule instruction ou construction, les **accolades { }** peuvent être omises.

Exemple : la fonction *pgcd()* ci-dessus peut être simplifiée de la manière suivante :

```
int pgcd (int a, int b)
{
   while (a != b)
      if (a < b)
         b = b - a;
      else
         if (b < a)
            a = a - b;
   return (a);
}
```

Un bloc peut être vide; cette latitude syntaxique est nécessaire dans certains contextes. Un bloc vide peut s'écrire d'une de deux manières :

```
{}    /* bloc sans instruction */
;     /* bloc d'une instruction, instruction sans opération */
```

Condition ( )

Une condition est une **expression** quelconque qui peut être évaluée comme *vraie ou fausse*.

Toute condition doit être placée entre **parenthèses ( )**.

2. Les constructions itératives

2.1. while

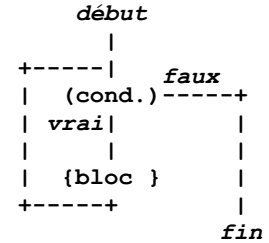
Syntaxe

```
while ( condition )
   { bloc }
```

La condition est une expression qui peut être évaluée comme vraie ou fausse.
Si le bloc est formé d'une seule instruction ou construction, les accolades peuvent être omises.

Interprétation

L'exécution du bloc se répète tant que la condition est vraie. L'évaluation de la condition est effectuée *avant* chaque exécution du bloc; si, dès le départ, la condition est fausse, le bloc est donc exécuté 0 fois.



Utilisation

La construction **`while`** est tout indiquée pour traiter les ***éléments d'un ensemble*** ... dont il est toujours avisé de prévoir qu'il pourrait être vide.

Pratiquement tous les programmes sont itératifs, en ce sens qu'ils répètent le même traitement sur des données d'entrée successives, données "lues" au terminal ou dans un fichier d'entrée principal. Il faut noter que, si l'ensemble traité contient *n* éléments, on devra effectuer *n+1* lectures (*n* éléments + *1* signal de fin).

Une *première lecture* doit précéder tout test de données – elle est donc effectuée au début du programme; on doit répéter l'opération de *lecture après* le traitement de chaque élément. La condition de répétition est la condition d'appartenance de l'élément courant à l'ensemble traité, ce qui s'exprime de différentes manières en fonction de la nature de l'ensemble. Le schéma de base habituel est donc celui-ci :

traitement d'un fichier :

```
première lecture;
while (!(signal de fin))
{
   opérations;
   lecture suivante;
}
```

parcours d'un tableau :

```
indice = 0;
while (indice < dimension)
{
   opérations;
   indice = indice + 1;
}
```

Exemple. Pour tester complètement la fonction *puiss()* donnée en exemple au chapitre 1, il est nécessaire de l'exécuter avec toutes les combinaisons de paramètres positifs, négatifs et nuls qu'autorisent les pré-conditions de l'algorithme. La fonction ci-dessous répète le test sur tous les couples de nombres introduits au clavier, jusqu'à ce que le premier de ces nombres soit 0.

```
#include <stdio.h>            /* fonctions de dialogue */
float puiss(float n,int p);         /* fonction testée */
void main (void)
{
  float r; int e;       /* racine exposant */
      /* 1ère. lecture : */
  printf("\nDonnez racine et exposant : "); scanf("%g%d",&r,&e);
  while (r != 0)
  {
        /* test de la fonction : */
    printf("\n %g exp %d = %g \n",r,e,puiss(r,e));
        /* lecture suivante : */
    printf("\nDonnez racine et exposant : "); scanf("%g%d",&r,&e);
  }
}
```

Le concept de ***lecture*** associée à une construction `while` est un concept abstrait. Il faut entendre sous ce terme toute opération fournissant au programme une nouvelle combinaison de données sur laquelle pourra s'exécuter une nouvelle fois le bloc d'instructions qui constitue le corps de la boucle.

Exemple. Voici un autre algorithme pour le calcul du PGCD de deux nombres.

```
DEBUT r ← reste de a/b                                  /* 1ère lecture */
        TANT QUE r ≠ 0
        EXECUTER    DEBUT a ← b
                          b ← r
                          r ← reste de a/b              /* lecture suivante */
                    FIN
        FIN-TANT
        ← b                             /* résultat = b */
FIN
```

Simplification de la construction

Lorsque l'opération de lecture est programmée sous la forme d'une expression (notamment, un appel de fonction) dont la valeur résultat est celle-là même que la construction `while` doit tester, on peut souvent simplifier le programme en incorporant au test de condition l'opération de lecture.

Exemple. Le programme suivant copie le flot *stdin* dans le flot *stdout*; si on en dévie la sortie vers un fichier, il permet de créer à partir du clavier un fichier de texte. La fonction de lecture *gets()* retourne 0 lorsqu'elle reçoit le signal de fin de fichier – cf. chapitre 3.

```
#include <stdio.h>
void main(void)          /* copie stdin dans stdout */
{
   char ligne[80+1];           /* 80 caractères + \0 */
   while (gets(&ligne) != 0)  puts(ligne);     /* !=0 facultatif */
}
```

Dans ce genre de simplification, on emploiera souvent une ***expression séquentielle***.

Exemple. Le programme ci-dessous répète le test de la fonction *puiss()* sur tous les couples de nombres introduits au clavier, jusqu'à ce que l'utilisateur introduise un signal de fin de fichier.

```
#include <stdio.h>            /* fonctions de dialogue */
float puiss(float n,int p);         /* fonction testée */

void main (void)
{
  float r; int e;        /* racine exposant */

  while ( printf("\nDonnez racine et exposant : "),/* expr.séqu. */
          scanf("%g%d",&r,&e) != EOF )   /* tant que non fin ... */
     printf("\n %g exp %d = %g \n",r,e,puiss(r,e));    /* tester */
}
```

Programmes hiérarchisés

Beaucoup de programmes sont *hiérarchisés*, en ce sens qu'ils ne se contentent pas de traiter séparément chaque élément d'un ensemble; ils contiennent également des traitements s'appliquant spécifiquement à l'ensemble et à des sous-ensembles d'éléments.

La structure de ces programmes est constituée de boucles *`while`* emboîtées l'une dans l'autre.

Illustrons ce schéma par l'exemple suivant. Lisant des relevés de fabrication par ateliers, on imprime les totaux de fabrication par usines et le total général (il s'agit d'une firme possédant plusieurs usines). Ce programme est hiérarchisé sur trois niveaux : traitement de l'ensemble "firme"*, traitement de chaque sous-ensemble* "usine"*, traitement de chaque élément* "atelier".

Chaque ligne du fichier d'entrée se présente comme ceci : `uaa total`
• u = *numéro d'usine (1 chiffre)* • aa = *numéro d'atelier dans l'usine (2 chiffres)* • blanc(s) • total •

```
#include <stdio.h>
void main (void)
{
  short int no_usine; short int no_atelier;
  int total_atelier;          /* données lues */
  int signal;  /* nbre. de données obtenues par scanf()
                  doit être 3 (<0 en fin de fichier) */
     /* 1ère. lecture : */
  signal = scanf("%1hd%2hd %d",&no_usine,&no_atelier,
                               &total_atelier);
  while (signal == 3)
  {    /* traitement d'une firme : */
    long int total_firme;       /* var. locale */
    total_firme = 0;
    while (signal == 3)
    {    /* traitement d'une usine : */
      long int total_usine;      /* var. locale */
      short int usine_traitee;   /* id. du ss-ens. */
      total_usine = 0;
      usine_traitee = no_usine;  /* mémo. indicatif */
      while (signal == 3 && no_usine == usine_traitee)
      {
           /* traitement d'un atelier : */
        total_usine = total_usine + total_atelier ;

           /* lecture suivante : */
        signal = scanf("%1hd%2hd %d",
                       &no_usine,&no_atelier,
                       &total_atelier);
      }
      printf("\nTotal usine %1hd = %ld",usine_traitee,
                                        total_usine);
      total_firme = total_firme + total_usine;
    }
    printf("\nTotal firme = %ld",total_firme);
  }
}
```

Le repérage des sous-ensembles successifs implique la mémorisation d'un indicatif (numéro) identifiant à tout moment le sous-ensemble en cours de traitement.

Le ***traitement d'un (sous-)ensemble*** comporte les trois phases suivantes :

- *début :* mémorisation de l'identifiant du sous-ensemble courant;
  production d'un titre;
  initialisation des valeurs cumulatives;
- *corps :* traitement en boucle des parties ou éléments;
- *fin :* production des valeurs cumulées (totaux, etc.) caractérisant l'ensemble.

Les ***variables*** (données cumulatives, titre, identifiant ...) caractérisant chaque sous-ensemble d'un même niveau de décomposition seront, en principe, déclarées localement à l'intérieur du bloc traitant les sous-ensembles de ce niveau.

2.2. for

Syntaxe

```
for ( init ; cond ; incr )
   { oper }
```

init, *cond*, *incr* sont trois expressions;
chaque expression est facultative, mais tous les points-virgules sont obligatoires.
cond est une expression qui peut être évaluée comme vraie ou fausse
(si elle est omise, l'exécution se déroule comme si l'on avait écrit une condition toujours vraie).
Si le bloc *oper* est formé d'une seule instruction ou construction, on peut omettre les accolades.

Interprétation

La construction **`for(`***`init;cond;incr`***`){`***`oper`***`}`** est équivalente à une construction `while` complétée de la façon suivante :

```
init;
while (cond)
{
   {oper}
    incr;
}
```

Comme ses équivalents dans les autres langages de programmation, la construction **`for`** a été conçue comme une formule plus compacte, susceptible de remplacer la construction `while` lorsque les pas de l'itération doivent être *comptés* :

- l'expression ***init*** affecte une valeur initiale au compteur – elle est omise si cette valeur est déjà établie;
- la condition ***cond*** vérifie que la valeur courante du compteur est comprise dans l'intervalle prévu – en pratique, puisque l'expression *init* garantit une initialisation correcte, la condition teste simplement que le compteur n'ait pas dépassé la fin de l'intervalle;
- l'expression ***incr*** incrémente le compteur ou, plus généralement, le fait évoluer vers la condition d'arrêt.

Utilisation

La construction `for` est employée pour piloter l'algorithme des **fonctions numériques itératives**, dont le nombre d'itérations dépend de la valeur d'un paramètre.

Exemple

```
float puiss(float n,int p)    /* élever n à la puissance p */
{
   float resultat;
   resultat = 1;
   for (;p>0;--p) resultat *= n;    /* p est déjà */
   for (;p<0;++p) resultat /= n;    /* initialisé */
   return (resultat);
}
```

Le compteur géré par la construction `for` sert souvent à indicer les éléments d'un **tableau**.

Les expressions de contrôle définissent un **intervalle** dans un vecteur : *{élément[m] ... élément[n]}*. Si, à chaque pas, l'incrément est positif, l'intervalle est exploré de gauche à droite ($m \rightarrow n$); si l'incrément est négatif, l'intervalle est exploré de droite à gauche ($m \leftarrow n$). Tous les éléments ne seront traités que dans le cas où la valeur absolue de l'incrément est 1.

Si l'on veut parcourir entièrement l'intervalle défini, les expressions de contrôle prennent une des formes suivantes (la condition de poursuite de l'itération porte sur la valeur de l'indice) :

```
for (i=m; i<=n; i=i+k)        for (i=n; i>=m; i=i-k)
```

Exemple : calcul des totaux de lignes et de colonnes d'un tableau.

```
.....
int montant[12][5];           /* 12 lignes x 5 colonnes */
long int total_ligne[12];
long int total_colonne[5];
int l;                        /* indice Ligne */
int c;                        /* indice Colonne */
.....
      /* initialisation des totaux de lignes : */
for (l=0; l<12; ++l)  total_ligne[l] = 0;
      /* initialisation des totaux de colonnes : */
for (c=0; c<5; ++c)  total_colonne[c] = 0;
      /* cumuls : */
for (l=0; l<12; ++l)
   for (c=0; c<5; ++c)
   {
      total_colonne[c] += montant[l][c];
      total_ligne[l] += montant[l][c];
   }
.....
```

On peut ne traiter qu'une partie de l'intervalle, en s'arrêtant dès la rencontre d'un élément servant de **délimiteur**, dont la position n'est pas connue a priori. La position occupée par le délimiteur n'appartient pas au sous-ensemble traité. Cas fréquent en C : traiter toutes les positions d'une **chaîne de caractères**, jusqu'au terminateur *\0* non compris.

Exemple. La fonction standard *strlen() – "string length"* indique la longueur du texte d'une chaîne de caractères, terminateur *\0* exclu. Le bloc d'instructions répétées dans la construction *for* est vide; cette construction est utilisée à seule fin de faire varier l'indice.

```
int strlen (char chaine[])    /* tableau de longueur quelconque */
{
   int i;
   for (i=0;chaine[i]; ++i)         /* <=> chaine[i]!=0 */
      ;              /* traitement vide */
   return (i);    /* i = nombre de caract. avant \0 */
}
```

La fonction *strlen()* ne vérifie pas qu'elle ne dépasse pas les limites du tableau de caractères; elle suppose que le terminateur *\0* est certainement présent ! Une telle programmation est hasardeuse.

Les expressions de contrôle de la boucle ***for*** devraient plutôt prendre une des formes suivantes, où l'ordre des tests dans l'expression de la condition tient compte du fait qu'aucun délimiteur pourrait ne se rencontrer dans l'intervalle exploré :

```
for (i=m; (i<=n && element[i]!=limite); i=i+k)
for (i=n; (i>=m && element[i]!=limite); i=i-k)
```

Exemple. Dans un texte, trouver les positions des premier et dernier caractères non blancs. Le bloc est vide et la construction *for* est utilisée à seule fin de faire varier les indices.

```
.....
char texte[80+1];       /* chaîne de 80 caractères maximum */
int prem; int dern;     /* indices limites */
.....
     /* strlen(texte) --> longueur de la chaîne, \0 exclu */
for (dern=strlen(texte)-1; dern>=0 && texte[dern]==' '; --dern)
   ;                    /* traitement vide */
for (prem=0; prem<=dern && texte[prem]==' '; ++prem)
   ;                    /* traitement vide */
.....
```

2.3. do while

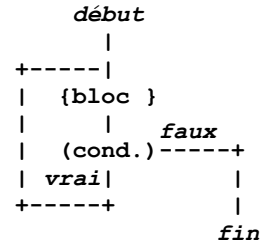
Syntaxe

```
do { bloc }
while ( condition ) ;
```

La condition est une expression qui peut être évaluée comme vraie ou fausse.
Si le bloc est formé d'une seule instruction ou construction, les accolades peuvent être omises.
La fin du bloc ne marquant pas la fin de la construction, celle-ci doit être clôturée par `;` .

Interprétation

L'exécution du bloc se répète tant que la condition est vraie. L'évaluation de la condition est effectuée *après* chaque exécution du bloc, ce qui garantit que le bloc soit exécuté au moins une fois.



Utilisation

Un cas typique d'emploi de la construction **`do while`** est la répétition d'une tentative d'opération sur un objet individuel jusqu'à ce que son résultat soit satisfaisant (une première tentative doit être effectuée avant que le résultat puisse être testé).

Exemple : introduction d'une réponse OUI/NON au clavier :

```
do
{
   printf("\noui OU non ? ");
   scanf ("%c%*c",&reponse);
/*  "%c" lit le caractère introduit et le range dans la variable
    "%*c" lit le caractère \n de fin de réponse sans le stocker
    ces deux formats ne peuvent pas être séparés par un espace  */
}
while ( !(reponse == 'o' || reponse == 'n') );
```

Remarquer dans cet exemple la formulation `while(!(`*condition de validité*`))` , généralement plus lisible que `while(`*conditions d'erreur*`)` , parce qu'elle explicite le but du traitement.

3. Les constructions alternatives

3.1. if

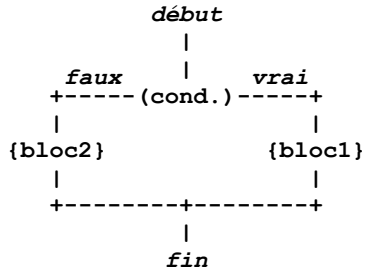
Syntaxe

```
if ( condition )   { bloc1 }
else               { bloc2 }
```

La condition est une expression qui peut être évaluée comme vraie ou fausse.
Si un bloc est formé d'une seule instruction ou construction, les accolades peuvent être omises.
Si le second bloc est vide, la construction peut être simplifiée en omettant `else {`*bloc2*`}`.

Interprétation

La condition est évaluée. Si elle est vraie, $bloc_1$ est exécuté; sinon, $bloc_2$ est exécuté.



Lorsque plusieurs constructions `if` sont emboîtées, chaque mot **`else`** apparaissant dans un bloc appartient à la dernière construction `if` du même bloc qui n'a pas encore de `else` correspondant.

Pour la lisibilité du texte, il est utile de décaler à une même distance de la marge gauche les mots `if` et `else` correspondants (voir le premier exemple ci-dessous).

Exemples

```
void adresse (char sexe,char etat_civil) /* impression d'adresse */
{
  char titre [3][14] = {"Madame ", "Mademoiselle ", "Monsieur "};
  if (sexe == 'M') printf (titre[2]);
  else if (etat_civil == 'C') printf (titre[1]);
       else                   printf (titre[0]);
}
```

```
int pgcd (int a, int b)  /* calcul du plus grand commun diviseur */
{
   if (a < b) return pgcd (a, b - a);
   if (a > b) return pgcd (b, a - b);
   if (a == b) return a;
}
```

```
int date_valide(int jj, int mm, int aaaa) /* contrôle d'une date */
{
  int jours_dans_mois[12] = {31,29,31,30,31,30,31,31,30,31,30,31};
  if (aaaa % 4 != 0) jours_dans_mois[2-1] = 28;
  return ( (mm >= 1 && mm <= 12)
         && (jj >= 1 && jj <= jours_dans_mois[mm-1]) );
}
```

Construction `if` ou expression conditionnelle ?

La puissance des expressions dans le langage C – particulièrement, des *expressions conditionnelles* – fait que les constructions `if` peuvent souvent être remplacées par de simples expressions.

Les exemples ci-dessus peuvent s'écrire comme ci-dessous.

```
void adresse (char sexe,char etat_civil) /* impression d'adresse */
{
  char titre [3][14] = {"Madame ", "Mademoiselle ", "Monsieur "};
  printf ( titre [(sexe=='M')? 2 : (etat_civil=='C')? 1 : 0] );
}
```

```
int pgcd (int a, int b)  /* calcul du plus grand commun diviseur */
{ return (a<b) ? pgcd(a,b-a) : (a>b) ? pgcd(b,a-b) : a ; }
```

```
int date_valide(int jj, int mm, int aaaa) /* contrôle d'une date */
{
  int jours_dans_mois[12] = {31,29,31,30,31,30,31,31,30,31,30,31};
  return ( mm >= 1 && mm <= 12 )
      && ( jj >= 1 && jj <= (mm == 2 ? aaaa % 4 == 0 ? 29 : 28
                                     : jours_dans_mois [mm-1]) );
}
```

3.2. switch

La construction **`switch`** ("aiguillage") comporte un bloc d'instructions à l'intérieur duquel sont définis plusieurs points d'entrée. L'exécution de la construction commence par évaluer une expression (souvent, une simple variable) et, en fonction de la valeur de cette expression, s'aiguille sur un des points d'entrée définis dans le bloc d'instructions.

Exemple : fonction renvoyant le nombre de jours du mois donné, ou 0 si le paramètre est invalide.

```
short int jours_mois (short int mois, short int annee)
{
  short int jours;
  switch (mois)   /* selon le numéro du mois ... */
  {
    case  1:
    case  3:
    case  5:
    case  7:
    case  8:
    case 10:
    case 12:      jours = 31;
                  break;      /* sortir de la construction */
    case  4:
    case  6:
    case  9:
    case 11:      jours = 30;
                  break;      /* sortir de la construction */
    case  2:      if (annee % 4 == 0) jours = 29; else jours = 28;
                  break;      /* sortir de la construction */
    default:      jours = 0;        /* autres cas : erreur */
  }
  return (jours);
}
```

Syntaxe et interprétation

```
switch ( expression )
  { bloc }
```

L'expression évaluée doit être d'un *type entier*. Elle doit être écrite entre parenthèses.

> Rappel utile. Un *caractère* est d'un type entier ... On peut donc tester un code alphabétique ...

Chaque point d'entrée à l'intérieur du bloc est étiqueté sous une des deux formes suivantes :

`case` *valeur* **:**	`default :`

Chaque étiquette **`case`** identifie le point d'entrée où commence l'exécution des instructions dans le cas où la valeur de l'expression testée est celle qui figure à la suite du mot `case`.

L'étiquette **`default`** est facultative. Elle identifie le point d'entrée où commence l'exécution des instructions lorsque la valeur de l'expression testée n'est pas une de celles qui sont reprises dans les étiquettes `case`.

> Chaque valeur de cas est représentée par une *expression constante* d'un type entier; il s'agira pratiquement toujours d'une simple constante littérale.
>
> Plusieurs valeurs peuvent conduire au même point d'entrée; en d'autres termes, différentes étiquettes de cas peuvent identifier un même point d'entrée.
>
> L'étiquette `default` peut être placée n'importe où, mais il est logique de la placer après les étiquettes de cas explicites.
>
> Il n'y aurait aucun sens à indiquer plusieurs fois la même étiquette.

```
break;
```

Normalement, le traitement d'un cas doit s'arrêter avant d'atteindre l'accolade de fin du bloc d'instructions. C'est ce que signifie l'instruction **`break`**,[26] qui ordonne de sortir immédiatement de la construction (dans l'exemple, on va exécuter *`return`*).

Conseil

Les différents cas sont disjoints et les opérations à exécuter dans chaque cas forment normalement des séquences disjointes; tel est l'effet de l'instruction *`break`*.

> L'exemple ci-dessous, produisant le même résultat que le précédent, l'obtient par des traitements non disjoints. Cette forme "sent l'astuce" et est beaucoup moins recommandable ... parce que (1) l'ordre de rédaction des cas et instructions ne peut pas être modifié; (2) l'utilité de cet ordre est rarement explicitée par un commentaire; (3) les valeurs entrant dans les calculs sont des valeurs déduites du raisonnement "astucieux" du programmeur plutôt que les valeurs "naturelles".

[26] `break` est une instruction *par soi-même*, utilisable à l'intérieur d'une construction `switch`, mais également dans d'autres contextes.

```
short int jours_mois (short int mois, short int annee)
/* fonction retournant le nombre de jours du mois donné
            retourne 0 si le numéro de mois est invalide */
{
  short int jours = 28; /* valeur posée au départ */
  switch (mois)       /* selon le numéro du mois ... */
  {
      /* 1°) mois de 31 jours */
    case  1: case  3: case  5:
    case  7: case  8: case 10: case 12:  ++ jours;
      /* 2°) mois de 30 jours */
    case  4: case  6: case  9: case 11:  ++ jours;
      /* 3°) février */
    case  2:      if (annee % 4 == 0)    ++ jours;  /* 29 jours */
            break;
    default:      jours = 0;        /* autres cas : erreur */
  }
  return (jours);
}
```

4. Les instructions de saut

Les instructions "de saut" rompent la séquence normale d'exécution d'une construction.

4.1. break – continue

Comme on vient de le voir, l'instruction **`break`** peut figurer dans le bloc d'instructions d'une construction *`switch`*. D'une manière analogue, l'instruction `break` peut figurer dans le bloc d'instructions d'une construction itérative *`while`*, *`for`* ou *`do while`*. Elle provoque un saut immédiat à l'instruction dont l'exécution doit suivre celle de *la plus petite construction* (`switch` ou boucle) qui l'entoure.

L'instruction **`continue`** peut figurer dans le bloc d'instructions d'une boucle *`while`*, *`for`* ou *`do while`*. Elle provoque le passage immédiat à l'itération suivante de *la plus petite construction itérative* qui l'entoure; contrairement à *`break`*, elle ne met pas fin au bouclage.

Exemple. Un tableau contient une chaîne de caractères représentant une expression à analyser. L'analyse doit s'arrêter (`break`) dès la rencontre du terminateur de chaîne *\0*; de plus, elle doit passer outre (`continue`) des blancs et autres caractères sans représentation graphique.

```
#include <ctype.h>      /* fonctions de test des caractères */
char expr[80+1];        /* long. maxi de l'expr. = 80 */
int posit;
.....
for (posit=0;posit<=80;++posit)
{
   if (expr[posit]=='\0') break;
   if (!isgraph(expr[posit])) continue;
   switch (expr[posit])
   { /* ... trait. des caractères pris en considération ... */ }
}
```

4.2. return *[expression]*

L'instruction **`return`** peut figurer à tout endroit du *corps d'une fonction*. Elle provoque une sortie immédiate de la fonction en cours et le retour à (l'opération suivante de) la fonction appelante.

> La fonction ***main()*** est censée appelée par le système de commande de l'ordinateur et retourne à ce système.

Dans la plupart des fonctions, l'instruction `return` a pour opérande l'*expression* dont la valeur est le résultat que la fonction rend à la fonction appelante.

> Si le corps d'une fonction ne contient pas l'instruction `return`, son exécution se termine à la prise en charge de l'accolade **`}`** finale.

4.3. goto *étiquette*

Toute instruction ou construction peut être précédée d'une étiquette. Une **étiquette** d'instruction est un identificateur (nom valide en C). Elle est suivie du signe deux-points.

étiquette **:**

> Pour pouvoir étiqueter l'accolade de fin d'un bloc, on doit la faire précéder d'une instruction vide :
>
> *étiquette* **`:`** **`;`** `}`
>
> La **portée** d'une étiquette s'étend sur toute la fonction où elle est déclarée.

L'instruction **`goto`** rompt la séquence normale d'exécution de la fonction où elle est rédigée, pour la poursuivre immédiatement au point de cette fonction identifié par l'étiquette référencée.

`goto` *étiquette* **`;`**

Remarques

> Les possibilités offertes par les expressions et constructions du langage C sont si riches, que l'instruction `goto` n'est pratiquement jamais employée.
>
> Une étiquette d'instruction n'est utilisable que dans une instruction `goto`. Il n'y a donc pas de possibilité de confusion entre une étiquette d'instruction et un identificateur de variable ou de fonction.

5. Fin d'exécution du programme : fonction *exit()*

La fonction *exit()* est définie dans le fichier d'en-tête standard ***stdlib.h*** de la manière suivante :

```
void exit(int status);
```

Rédigé dans n'importe quelle fonction, l'appel de la fonction ***exit(****status****)*** clôture immédiatement l'exécution du programme, en renvoyant au système de commande de l'ordinateur la valeur *status*. (Dans la fonction *main()*, l'intruction `return` *status* provoque le même effet.)

> Par convention, dans le système UNIX, `exit(0)` signifie que l'exécution s'est déroulée normalement et le renvoi de toute autre valeur, qu'il s'est produit des conditions particulières (au sujet desquelles l'information nécessaire doit être fournie par la documentation du programme).
>
> On peut donner en guise de paramètre l'une des constantes EXIT_SUCCESS ou EXIT_FAILURE. Ces constantes sont définies dans le fichier d'en-tête <*stddef.h*>.

6. La récursivité

La **récursivité** est un autre mécanisme de composition d'opérations. On qualifie de récursif un algorithme qui se rappelle lui-même. En C, l'algorithme rappelé doit nécessairement avoir la forme d'une fonction.

> On a donné au chapitre 1 un algorithme récursif pour le calcul du plus grand commun diviseur de deux nombres entiers.

La **recherche dichotomique** ("binary search") d'un élément dans un tableau ***ordonné*** est un algorithme classique, car il est très performant. En effet, le nombre de comparaisons effectuées pour trouver un élément dans un tableau de N éléments est $\leq \lceil \log_2(N+1) \rceil$.[27] Dans un tableau de ± 1000 éléments, le nombre d'éléments testés sera au maximum de l'ordre d'une dizaine. (Si la recherche se faisait en parcourant séquentiellement le tableau, ce nombre serait N+1.)

L'algorithme est celui-ci :

> *chercher l'élément x dans l'intervalle [0..N-1] du tableau :*
>
> – comparer x avec l'élément situé au milieu de l'intervalle;
> – si x est plus petit, le rechercher dans le demi-intervalle de gauche;
> – si x est plus grand, le rechercher dans le demi-intervalle de droite;
>
> – continuer en découpant toujours des demi-intervalles jusqu'à ce que
> – soit on trouve,
> – soit l'intervalle est devenu vide (contient 0 élément).

chercher.c (version récursive)

```
int chercher(int x, int t[], int gauche, int droite)
   /* recherche dichotomique dans un intervalle ordonné */
   /* paramètres : x = nombre recherché
                   t[] = tableau
                         - la dimension peut être laissée inconnue,
                           puisqu'on indique les bornes
                   gauche, droite = indices délimitant l'intervalle
      retour : indice de l'élément trouvé ou -1 en cas d'échec */
```

[27] Il serait plus exact de parler du nombre d'éléments testés plutôt que du nombre de comparaisons effectuées.

```
{
   if (gauche > droite)       /* si l'intervalle est vide */
      return (-1);                 /* --> non trouvé */
   else
   {
      int milieu;                          /* indice */
      milieu = (gauche + droite) / 2;
      if (x < t[milieu])      /* chercher dans la moitié gauche */
         return chercher(x, t, gauche, milieu - 1);
      if (x > t[milieu])      /* chercher dans la moitié droite */
         return chercher(x, t, milieu + 1, droite);
      return (milieu);              /* x == t[milieu] --> trouvé */
   }
}
```

```
#include <stdio.h>
#include "chercher.c"

void main(void)         /* test de la fonction chercher() */
{
   int t[7] = {1,2,4,8,16,32,64}; /* exemple de tableau ordonné */
   int n;
   printf("Donnez un nombre entier : "); scanf("%d",&n);
   printf("\n se trouve à la position %d", chercher(n,t,0,7-1));
}
```

Pour que l'exécution d'un algorithme récursif ne se répète pas à l'infini,

1) tout appel récursif doit être situé à l'intérieur d'une alternative, dans laquelle il existe une "issue" non récursive

la fonction de recherche dichotomique comporte deux issues non récursives :
- si le test d'égalité est satisfait : issue "trouvé";
- si gauche > droite (intervalle vide) : issue "non trouvé".

2) la répétition de l'appel récursif doit faire évoluer les données vers une condition d'issue non récursive

dans une recherche dichotomique, chaque nouvel appel de la fonction réduit l'intervalle à examiner; on finira donc par tomber sur une des deux issues non récursives.

Elimination de la récursivité

On peut toujours éliminer la récursivité d'un programme en le transformant. L'exécution de la version non récursive est plus performante, car elle évite la répétition des opérations assez complexes d'appel et de retour de fonctions. Mais elle est habituellement bien plus difficile à concevoir et bien moins compréhensible.

Cependant, lorsque, comme dans l'exemple ci-dessus, *l'appel récursif est la dernière opération exécutée dans le corps de la fonction* (l'exécution de cet appel n'est suivie que de `return`), la transformation du programme est aisée. Il s'agit de substituer à la répétition des appels récursifs un traitement en boucle. Le programme transformé épouse le schéma suivant :

TANT QUE NON (issue non récursive)
EXECUTER simulation de l'appel récursif
FIN-TANT
traitement final d'issue non récursive

La simulation d'un appel récursif consiste simplement à *donner de nouvelles valeurs aux paramètres de la fonction*.

chercher.c (version itérative)

```
int chercher(int x, int t[], int gauche, int droite)
   /* recherche dichotomique dans un intervalle ordonné */
   /* paramètres : x = nombre recherché
                   t[] = tableau
                   gauche, droite = indices délimitant l'intervalle
      retour : indice de l'élément trouvé ou -1 en cas d'échec */
{
   while (!(gauche > droite))   /* tant qu'intervalle non vide */
   {                            /*     procéder à la recherche */
      int milieu;                        /* indice */
      milieu = (gauche + droite) / 2;
      if (x == t[milieu]) return (milieu);     /* --> trouvé */
      if (x < t[milieu]) droite = milieu - 1;  /* moitié gauche */
      if (x > t[milieu]) gauche = milieu + 1;  /* moitié droite */
   }
   return (-1);          /* intervalle vide --> non trouvé */
}
```

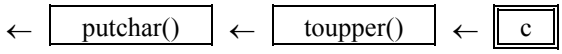
7. Programmation par fonctions

7.1. Méthodes de composition de fonctions

Un programme C est un ensemble de fonctions, une *composition de fonctions*. Comment se combinent ces fonctions ?

Supposons que l'on veuille afficher au moyen de la fonction putchar() *le caractère* c *converti en majuscule (s'il s'agit d'une lettre minuscule) par la fonction* toupper(). *[Rappel : le résultat de la fonction* putchar() *est un signal confirmant ou infirmant la bonne exécution de l'affichage.]*

Le graphe ci-dessous représente les affectations de paramètres et résultats, le *flux* des données, dans ce problème :



Deux observations sont à faire :

– la *chronologie d'exécution* des opérations doit suivre le sens (de droite à gauche) du flux des données;
– la seule chose qui "intéresse" le programme est le *résultat final*, figuré par la flèche la plus à gauche, et il peut "ignorer" *comment* ce résultat est obtenu, c'est-à-dire tout ce qui est représenté à droite de cette flèche.

Il existe trois **méthodes de composition de fonctions**, qui ne sont pas toujours interchangeables.

Composition séquentielle

Chaque fonction peut être appelée par une instruction-expression dont le résultat est mémorisé dans une variable de liaison; la séquence des instructions doit refléter l'ordre chronologique :

```
var = toupper(c); signal = putchar(var);
```

Cette méthode de composition présente l'inconvénient de nécessiter des variables intermédiaires (`var`). En outre, une composition d'instructions ne peut pas figurer à l'intérieur d'une expression.

Le second inconvénient peut être supprimé en adoptant la forme d'une **expression séquentielle** :

```
signal = ( var = toupper(c), putchar(var) );
```

Composition par emboîtement

On peut rédiger une expression formée d'appels de fonctions emboîtés :

```
signal = putchar(toupper(c));
```

Cette méthode fait l'économie des variables intermédiaires. Lorsqu'elle est possible, on la préfèrera donc à la composition séquentielle.

On remarquera que l'ordre de rédaction des appels de fonctions emboîtés dans une expression est l'inverse de l'ordre chronologique de leur exécution.

Composition cachée ("encapsulation")

Avec les deux méthodes ci-dessus, le programme doit connaître le détail du fonctionnement de l'opération "afficher un caractère en majuscule" qu'il demande, c'est-à-dire la liste des fonctions composantes *toupper()* et *putchar()* et le flux de données interne à l'opération "afficher...".

Il est préférable de ***cacher*** ou *"enfermer"* (*"encapsulate"*, en anglais) les détails du fonctionnement, et de proposer au programme utilisateur une fonction unique.

```
int affiche_majusc (int caract)
{  return putchar(toupper(caract));  }

signal = affiche_majusc(c);
```

7.2. Critères de décomposition en fonctions

Un programme C est une composition de fonctions, dont certaines préexistent et d'autres sont à inventer.

Pour concevoir et structurer son programme, le programmeur applique donc une double démarche. Par une **méthode ascendante** (*"bottom up"*), il cherche à créer des opérateurs (fonctions) plus adaptés à la résolution de son problème que ceux dont il dispose a priori; il crée ainsi, en quelque sorte, une *machine abstraite*. Par une **méthode descendante** (*"top down"*), il cherche à décomposer le problème en problèmes moins complexes, plus aisés à programmer ... sur sa machine abstraite.

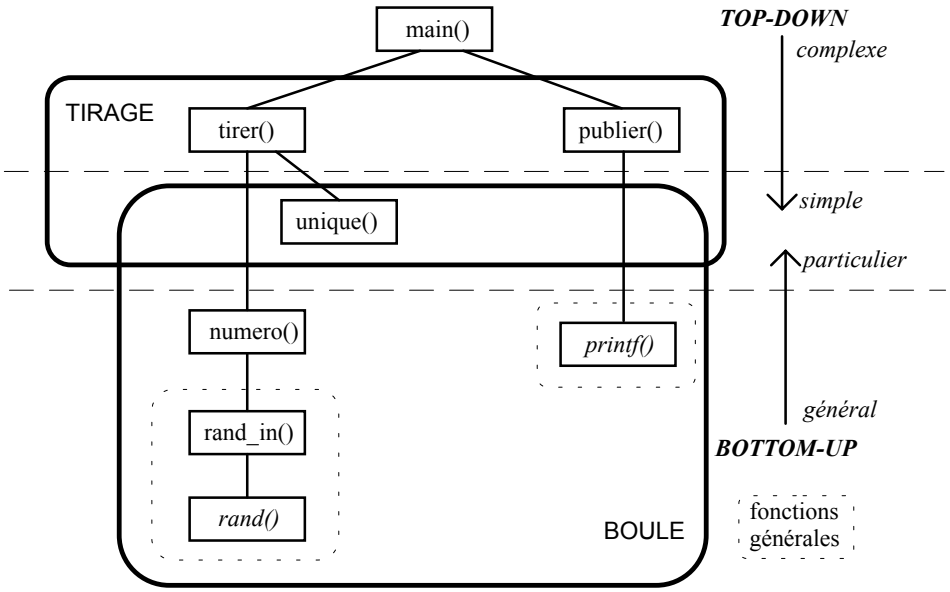
In top-down design we say "this problem is too complex for me; I shall dissect it into a number of smaller problems, and those into smaller problems still, until I have a set of problems simple enough to solve". In bottom-up design we say instead "this machine is not well suited to my problem, because the elementary operations are too elementary; I shall therefore use the elementary operations to create a more powerful machine, and the operations of that machine to create one still more powerful, until I have a machine on which my problem can be easily solved".

(...) For example, if we were forced to write a set of mathematical programs, involving manipulation of matrices, we might reasonably begin by designing a new machine which was capable of arithmetic operations on matrices. We might specify a new data type, the matrix, and a set of operations MATRADD, MATRSUB, MATRMUL and MATRDIV. (...) Such components must be thought of by their user, the top-down designer, as elementary components of the machine he is using and therefore not susceptible to internal examination.

M.A. JACKSON, ***Principles of Program Design***; London, Academic Press, 1975.

Le programme ci-dessous va nous servir à illustrer cette double démarche et les critères qu'elle utilise pour identifier les fonctions à créer.

Ce programme simule le tirage du ***loto***. Il manipule deux types d'**objets** : *boule* et *tirage*. Un objet *boule* est représenté par un nombre entier aléatoire pris dans l'intervalle [1,42]; l'objet *tirage* est représenté par un tableau de 6 numéros de base + 1 numéro complémentaire, tous différents. Le programme sera un algorithme combinant des opérations (fonctions) sur ces deux types d'objets.[28] Dès le départ, le programmeur pressent que les opérations sur une *boule* doivent être élaborées (méthode ascendante) sur la base de la fonction standard `rand()` de génération de nombres pseudo-aléatoires; quant aux opérations sur l'objet *tirage*, d'évidence elles se décomposent (méthode descendante) en itérations sur les éléments du tableau.



[28] Cette discussion méthodologique relève d'une inspiration proche de celle de la *programmation par objets*.

Conception ascendante

La fonction de tirage d'un numero() *de boule consiste à ramener dans l'intervalle [1,42] le nombre pseudo-aléatoire fourni par la fonction standard* rand(). *L'algorithme demeurerait inchangé pour n'importe quel intervalle [min,max]. Ici apparaît une possibilité de généralisation : créer une fonction* rand_in(min,max)*, qui pourra être réutilisée dans un autre contexte que celui de ce jeu de loto. Le flux de données est le suivant :* ←`numero()`←`rand_in()`←`rand()`.

• L'identification des fonctions *numero()* et *rand_in()* résulte du raisonnement suivant : le flux de données procède du général au particulier; chacune de ses étapes opère une **spécialisation** ou **sous-typage** : à chaque étape, le type ou domaine de valeurs possibles se réduit à un sous-ensemble du précédent. Dans ce style de raisonnement, on découvre fréquemment des occasions de **généraliser la solution**, en créant des fonctions de niveau intermédiaire, comme ici la fonction *rand_in()*.

• La démarche de spécialisation ou sous-typage est une forme particulière de celle qui consiste à distinguer des **états intermédiaires** dans un flux de données (ex.: ← *mis en page* ← *trié* ← *extrait* ← *lu*).

Conception descendante

Au premier niveau, le programme main() *sera décomposé en deux fonctions* effectuer() *et* publier() *le tirage.*

• Critère d'identification de ces fonctions : chacune est une **opération indépendante sur un objet** *tirage* (au même titre que la fonction *numero()* sur l'objet *boule*). On peut imaginer d'autres fonctions indépendantes sur l'objet *tirage* comme, par exemple, le tri des 6 numéros de base. Bien que découvertes par une autre sorte de démarche, ces fonctions respectent le critère évoqué au paragraphe précédent : chacune entraîne un **changement d'état** de l'objet.

• Par ailleurs, dans de très nombreux programmes, on peut distinguer **différents objets** (par exemple, un fichier d'entrée et une liste imprimée). Différentes fonctions seront identifiées pour chaque objet (exemples : lire le fichier et trier le fichier, mettre la liste en page et imprimer la liste).

• Autre critère conduisant à la même décomposition : les fonctions *effectuer()* et *publier()* correspondent à deux **phases du déroulement** du programme, phases nettement séparables. Normalement, les fonctions identifiées à l'aide de ce critère ne rendent pas de valeur résultat; elles constituent ce que le langage PASCAL appelle des ***procédures*** (fonctions sans retour de valeur).

L'opération effectuer() *exécute trois boucles imbriquées l'une dans l'autre : [1] boucle garnissant les 7 positions du tableau, [2] répétition du tirage d'un numéro si la fonction* numero() *a renvoyé un numéro déjà tiré, [3] comparaison de ce nouveau numéro avec chacun des précédents pour savoir s'il a déjà été tiré. Afin de rendre la programmation moins complexe, on décide d'isoler la boucle [3] dans une fonction de test* unique().

• Isoler une partie du traitement dans une fonction distincte aide le programmeur à dominer intellectuellement le programme. A cette fin, la meilleure méthode consiste à **désimbriquer les constructions algorithmiques**, itératives (`while`, `for`, `do`), alternatives (`if`, `switch`) et récursives. (Dans notre programme, la fonction *effectuer()* comportera seulement deux boucles imbriquées, plutôt que trois.)

On peut aussi considérer que la fonction unique() *complète la chaîne des fonctions appliquées à l'objet* boule.

Exemple

Rappel. Lorsqu'on passe un *tableau en paramètre* à une fonction, celle-ci reçoit en réalité l'*adresse* de ce tableau (et non pas une copie du contenu du tableau). En conséquence, si la fonction appelée modifie le contenu du tableau, cette modification altère le contenu du paramètre effectif dans la fonction appelante.

loto.c

```
      /***** DEFINITION DES FONCTIONS STANDARDS *****/

#include <stdio.h>      /* printf() */
#include <stdlib.h>     /* rand(), srand() */
#include <time.h>       /* time() */

      /***** FONCTIONS MANIPULANT L'OBJET boule *****/

int rand_in (int min, int max)
   /** ramener dans l'intervalle [min,max]
       le nombre aléatoire reçu de rand() **/
{  return ( min + rand() % (1 + max - min) );  }

int numero (void)
   /** tirer une boule = nombre aléatoire dans [1,42] **/
{  return ( rand_in(1,42) );  }

      /***** FONCTION UTILISANT LES 2 OBJETS boule ET tirage *****/

int unique (int boule, int tirage[6+1], int position)
          /* position de la boule testée -
             cette boule est déjà ou non placée dans le tableau
             renvoie le n° de boule si OK; sinon 0, càd. FAUX */
{
   int i;
   for (i=0; i<position; ++i) if (boule == tirage[i]) return (0);
   return (boule);
}

      /***** FONCTIONS MANIPULANT L'OBJET tirage *****/

void effectuer (int tirage[6+1])    /** effectuer le tirage **/
{
   int i;
   srand(time(0));      /* initialiser le générateur rand() */
   for (i=0; i<7; ++i)  /* tirer 7 numéros */
      do                            /* répéter la tentative */
         tirage[i] = numero();        /* si le numéro */
      while (! unique(tirage[i],tirage,i)); /* a déjà été tiré */
}

void publier (int tirage[6+1])      /** publier le résultat **/
{
   int i;
   for (i=0; i<7; ++i) printf("%d  ",tirage[i]);
}
```

```
            /***** PROGRAMME *****/

      void main (void)
      {
         int tirage[6+1];           /* tableau résultat du tirage */
         effectuer(tirage); publier(tirage);
      }
```

Exercices

while

1.

Programmer une fonction de calcul du plus grand commun diviseur de deux nombres entiers positifs utilisant l'algorithme suivant :

```
PGCD(a,b)
r ← reste de a/b
TANT QUE r ≠ 0
EXECUTER      a ← b,  b ← r
              r ← reste de a/b
FIN-TANT
← b
```

Ecrire un programme pour tester cette fonction, tel qu'un nombre indéterminé de tests puissent être exécutés successivement. Ne pas oublier de définir la convention d'arrêt de la répétition (nombre 0, signal de fin de fichier ... ?).

2.

Lorsqu'un utilisateur répond à une demande d'un programme, il est toujours susceptible d'introduire au clavier un nombre quelconque de caractères (de 0 à n), et sa réponse est toujours clôturée par le caractère `\n` de fin de réponse (touche ENTER).

Créer une fonction *reponse()* qui renvoie à la fonction appelante le premier caractère de la réponse, mais qui lit néanmoins tous les caractères introduits, jusqu'au caractère `\n` compris. Puisque le nombre de caractères introduits est quelconque, il est, en toute rigueur, impossible de définir un tableau de taille suffisante pour recevoir le texte complet de la réponse; les éventuels caractères après le premier seront donc "oubliés". (Remarque : il est possible que le premier caractère soit déjà le terminateur `\n` ...) – Quelle fonction de lecture est la plus adaptée à la solution de ce problème ?

do while

3.

Corriger l'algorithme de demande et réception d'une réponse *oui/non* fourni en exemple à la page 5-11, de telle sorte qu'il utilise la fonction *reponse()*.

do while – if

4.

Programmer une fonction calculant, avec une précision ε donnée (par exemple, 0.0001), la racine carrée positive d'un nombre réel `a` strictement positif. La solution est le premier r_n de la suite :

$r_1 = a, \quad r_n = 1/2\ (r_{n-1} + a/r_{n-1}), \quad n > 1$ tel que $|r_n - r_{n-1}| \leq \varepsilon$

(La fonction doit appeler une autre fonction qui renvoie la valeur absolue d'un nombre réel.)

for – if

5.

Programmer une seule fonction contenant la déclaration d'un vecteur de 20 entiers et exécutant successivement sur ce tableau les trois traitements suivants.

Garnir toutes les positions du vecteur en appelant la fonction standard *rand()* de génération de nombres pseudo-aléatoires.

Trier – c'est-à-dire ordonner – les éléments du tableau du plus petit au plus grand. Utiliser l'algorithme du *tri par sélection* :
- sélectionner dans l'intervalle *t[0] ... t[19]* le plus petit élément et le permuter avec l'élément *t[0]*;
- sélectionner dans l'intervalle *t[1] ... t[19]* le plus petit élément et le permuter avec l'élément *t[1]*;
- continuer de la même manière pour les intervalles *t[2] ... t[19]* jusque *t[18] ... t[19]*.

Afficher le résultat sur deux colonnes : la première colonne liste les nombres en ordre croissant, la seconde colonne les liste en ordre décroissant. Programmer cela en une seule boucle `for`.

switch – while

6.

Créer un programme unique pour tester les trois fonctions mathématiques *puiss()*, *pgcd()*, *ppcm()* du chapitre 1.

Ce programme affiche à l'écran le "menu" suivant :

```
PROGRAMME DE TEST

      1     puiss()
      2     pgcd()
      3     ppcm()

Quelle fonction désirez-vous tester ? _
```

Suivant le numéro introduit en réponse, exécuter *un* test de la fonction correspondante. En cas de réponse différente, ne rien faire.

Ce programme comprend les fonctions suivantes :

- *puiss()*, *pgcd()*, *ppcm()* – les prendre dans le texte du chapitre 1;
- une fonction *menu()* qui affiche le texte du menu et demande le choix de l'utilisateur;
- une fonction *tests()* qui, suivant la réponse reçue, effectue un des trois tests possibles (le test d'une fonction consiste à demander deux nombres et à afficher le résultat calculé);
- la fonction *main()*, assurant qu'un nouvel accès au menu soit effectué après chaque test – l'exécution s'arrête lorsque l'utilisateur donne en réponse le nombre 0.

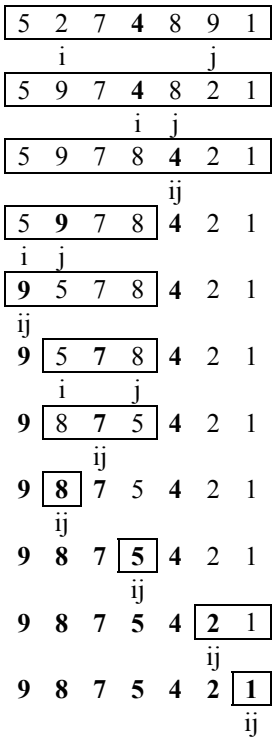
Récursivité

7.

Ecrire une fonction de **tri rapide** (*"quick sort"*) d'un tableau de nombres entiers, *ne contenant pas de valeurs en double*.

L'idée consiste à trouver la place définitive du *pivot*, valeur occupant initialement la case médiane du tableau. Lorsque ceci est fait, on réapplique récursivement l'algorithme sur les deux parties du tableau situées respectivement à gauche et à droite de la nouvelle position du pivot.

L'algorithme pour le placement du pivot est le suivant (pour un tri en ordre *décroissant*) :
– partant de l'extrémité gauche de l'intervalle examiné, trouver la position *i* du premier élément inférieur ou égal au pivot;
– partant de l'extrémité droite de l'intervalle examiné, trouver la position *j* du premier élément supérieur ou égal au pivot;
– si $i < j$, permuter les éléments correspondants;
– continuer ainsi jusqu'à ce que $i = j$ – le pivot occupe alors son emplacement définitif.



8.

Soit un produit de quatre facteurs $T_1 * T_2 * T_3 * T_4$; les multiplications peuvent être effectuées en suivant différents ordres, par exemple : $(T_1 * (T_2 * (T_3 * T_4)))$, $((T_1 * T_2) * (T_3 * T_4))$, etc. Rédigez la version procédurale d'une fonction récursive qui calcule le nombre d'ordres d'exécution différents possibles pour un produit de *q* facteurs ($q \geq 1$).

Illustration :

	ordres possibles	*méthodes*	***nombre de solutions***
pour 1 facteur	$f_1(1) = T_1$		**1**
pour 2 facteurs	$f_1(2) = (T_1 * T_2)$	$f_1(1) * f_1(1)$	**1**
pour 3 facteurs	$f_1(3) = (T_1 * (T_2 * T_3))$	$f_1(1) * f_1(2)$	
	$f_2(3) = ((T_1 * T_2) * T_3)$	$f_1(2) * f_1(1)$	**2**
pour 4 facteurs	$f_1(4) = (T_1 * (T_2 * (T_3 * T_4)))$	$f_1(1) * f_1(3)$	
	$f_2(4) = (T_1 * ((T_2 * T_3) * T_4))$	$f_1(1) * f_2(3)$	
	$f_3(4) = ((T_1 * T_2) * (T_3 * T_4))$	$f_1(2) * f_1(2)$	
	$f_4(4) = ((T_1 * (T_2 * T_3)) * T_4)$	$f_1(3) * f_1(1)$	
	$f_5(4) = (((T_1 * T_2) * T_3) * T_4)$	$f_2(3) * f_1(1)$	**5**
pour 5 facteurs	$f_1(5) =$	$f_1(1) * f_1(4)$	
	$f_2(5) =$	$f_1(1) * f_2(4)$	
	$f_3(5) =$	$f_1(1) * f_3(4)$	
	$f_4(5) =$	$f_1(1) * f_4(4)$	
	$f_5(5) =$	$f_1(1) * f_5(4)$	
	$f_6(5) =$	$f_1(2) * f_1(3)$	
	$f_7(5) =$	$f_1(2) * f_2(3)$	
	$f_8(5) =$	$f_1(3) * f_1(2)$	
	$f_9(5) =$	$f_2(3) * f_1(2)$	
	$f_{10}(5) =$	$f_1(4) * f_1(1)$	
	$f_{11}(5) =$	$f_2(4) * f_1(1)$	
	$f_{12}(5) =$	$f_3(4) * f_1(1)$	
	$f_{13}(5) =$	$f_4(4) * f_1(1)$	
	$f_{14}(5) =$	$f_5(4) * f_1(1)$	**14**

Composition de fonctions

9.

Le principe de l'algorithme de **tri par insertion** dans un tableau est le suivant :

1) les éléments du tableau sont ordonnés dès leur insertion dans le tableau;
2) si le nouvel élément doit s'intercaler à l'intérieur de la suite déjà garnie, les éléments de fin de cette suite sont décalés.

(QEJZG): → **Q** → **E**Q → E**J**Q → EJQ**Z** → E**G**JQZ

Programmer cet algorithme pour traiter un tableau d'entiers `long int` de dimension quelconque; les nombres doivent être rangés en ordre croissant. Réaliser l'algorithme au moyen de trois fonctions *localiser()*, *decaler()*, *inserer()*.

a) La fonction *localiser()*, en examinant la suite de nombres déjà constituée, donne l'indice de la position à laquelle le nouveau nombre doit être inséré (ou, le cas échéant, à laquelle il se trouve déjà rangé). Cette fonction est une variante légèrement modifiée de la version itérative de la fonction *chercher()* de recherche dichotomique.
b) La fonction *decaler()* décale vers la droite la fin de la suite de nombres déjà constituée.
c) La fonction *inserer()* insère à la position voulue le nouveau nombre, sauf si ce nombre existe déjà dans le tableau.

Une de ces fonctions – laquelle ? – peut cacher les deux autres.

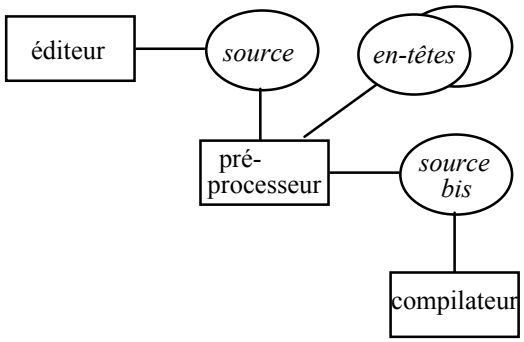
Remarques

1) Distinguer le nombre d'éléments présents dans le tableau et le nombre d'emplacements – c'est-à-dire la dimension – du tableau. Au retour, le programme appelant doit recevoir l'indication du nouveau nombre d'éléments présents; si ce nombre est supérieur à la dimension du tableau, cela signifie que l'insertion demandée n'a pu se faire, le tableau étant plein.
2) Les paramètres des différentes fonctions peuvent être l'élément à insérer, le tableau, des indications de dimension ou position ...

Ecrire un programme de test, demandant à la fonction standard *rand()* les valeurs à ranger dans le tableau.

Chapitre 6. Le pré-processeur

Le **pré-processeur** transforme le texte source fourni par le programmeur, conformément aux ***directives*** de transformation que celui-ci y a incorporées, et transmet le texte modifié au compilateur.



Toute ligne du texte source dont le premier caractère non blanc est # porte une (et une seule) directive pour le pré-processeur. Le caractère # initial peut être librement précédé et suivi de blancs.

1. Mise en page du programme

Les premières transformations subies par le texte source sont des opérations de mise en page :

1) *concaténation de lignes* :
si le dernier caractère non blanc d'une ligne est le caractère ****,
ce caractère est supprimé et le texte de la ligne suivante est accolé à cet endroit
2) *suppression des commentaires* (textes entre **`/* */`**)

La dernière opération du pré-processeur est la suivante :

3) *concaténation des chaînes de caractères* adjacentes.

Exemple

texte fourni par le programmeur :

```
int main(void)   /* démonstration */
{ puts ("Miroir\
, dis-moi : " "suis-je la plus belle ?"); }
```

première transformation :

```
int main(void)
{ puts ("Miroir, dis-moi : " "suis-je la plus belle ?"); }
```

texte livré au compilateur :

```
int main(void)
{ puts ("Miroir, dis-moi : suis-je la plus belle ?"); }
```

2. Les fichiers d'en-tête *.h*

La directive **`#include`** ordonne au pré-processeur d'inclure dans le texte source transmis au compilateur le texte du fichier qu'elle désigne. Ce fichier est appelé un **fichier d'en-tête** et, habituellement, possède un nom de la forme *nom*`.h` ("header").

La directive `#include` existe dans deux formats :

`#include <`*fichier*`>`	le fichier est cherché dans les répertoires "standards" de l'environnement de développement en C;
`#include "`*fichier*`"`	le fichier est cherché conformément à sa désignation puis, s'il n'est pas trouvé, dans les répertoires standards.

Exemples :

```
   /* bibliothèque des fonctions mathématiques standards */
#include <math.h>
   /* bibl. de fonctions additionnelles, non standards */
#include "math_plus.h"
```

3. Les macro-définitions

3.1. Macro-définitions constantes

Au lieu d'écrire en de multiples endroits du programme une même valeur constante, on peut la remplacer partout par un nom plus "parlant". Cette technique est très fréquemment employée pour les valeurs codées.

Exemples :

```
/* codification des sexes : 1 = homme, 2 = femme */
# define HOMME    1
# define FEMME    2
/* codification de la réponse oui/non */
#define OUI       'o'
#define NON       'n'
```

Une macro-définition constante est une ligne de la forme suivante :

`# define` *nom* *texte de remplacement*

Le *nom* doit être un identificateur valide en C. Les programmeurs ont l'habitude d'écrire en majuscules un nom de macro-définition; c'est devenu une convention de fait.

Le *texte* de remplacement est un texte quelconque. Il doit être précédé d'au moins un espace et s'étend jusqu'à la fin de la ligne. (S'il doit s'étendre sur plusieurs lignes, il suffit d'écrire le caractère \ en fin de chaque ligne coupée.)

Le texte de remplacement peut être "vide", c'est-à-dire inexistant. Ceci se présente notamment dans certains cas de compilation conditionnelle (cf. infra).

Le premier exemple ci-dessus (codification des sexes) montre un premier intérêt de la technique : des noms sont beaucoup plus parlants que des valeurs codées.

Le deuxième exemple (codification OUI/NON) montre un autre intérêt de la technique : si la codification possède des variantes ou vient à être modifiée, seule la ligne de définition `#define` doit être adaptée, tandis que le nom répété à l'intérieur du programme reste inchangé.

Exemple :
```
/* codification oui/non anglaise */
#define OUI       'y'
#define NON       'n'
```

La même technique est également souvent employée pour indiquer une limite ou une grandeur inconnue au moment de l'écriture du programme, ou susceptible de changer.

Exemple :
```
/* dimension d'un tableau */
# define N  42
...
int matrice_carree[N][N]
...
for (i=0;i<N;++i)
...
```

texte livré au compilateur :
```
...
int matrice_carree[42][42]
...
for (i=0;i<42;++i)
...
```

REMARQUE. Des macro-définitions constantes pourraient se substituer aux mots clés du langage ... ce qui définirait un langage de pseudo-code compilable ...

Exemple :
```
/* pseudo-langage minimum */
# define  PROGRAMME   void main (void)
# define  CARACTERE   char
# define  ENTIER      long int
# define  REEL        float
# define  DEBUT       {
# define  FIN         }
# define  SI          if (
# define  ALORS       ) {
# define  SINON       } else {
# define  FIN_SI      }
# define  TANT_QUE    while (
# define  EXECUTER    ) {
# define  FIN_TANT    }
# define  RETOURNER   return
# define  OU          ||
# define  ET          &&
# define  NON         !
# define  ENTRER      scanf
# define  AFFICHER    printf
```

3.2. Macro-définitions paramétrables

Une macro-définition peut être paramétrable.

`# define` *nom*`(`*p1*`,`*p2*`,...)` *texte de remplacement utilisant les paramètres*

Le *nom* de la macro-définition est suivi d'une liste de paramètres variables.

Chaque *paramètre* est lui-même représenté par un nom. La liste doit être collée au nom, *sans espace avant la parenthèse ouvrante*.[29]

Le *texte* de remplacement est soumis aux même règles que pour une macro-définition constante. Dans le cas d'une macro-définition paramétrable, ce texte fait normalement usage des paramètres.

Pour éviter des problèmes dus aux niveaux de priorité dans les expressions, il est vivement recommandé de placer entre parenthèses chaque mention de paramètre apparaissant à l'intérieur du texte de remplacement.

Exemple :
```
#define PERMUTER(a,b)   ((a) ^= (b) ^= (a) ^= (b))
...
unsigned long int n[33]; char c[121];
...
... PERMUTER(n[i],n[j]);
... PERMUTER(c[k],c[121-k]);
```

le texte livré par le pré-processeur au compilateur devient :
```
...
unsigned long int n[33]; char c[121];
...
... ((n[i]) ^= (n[j]) ^= (n[i]) ^= (n[j]));
... ((c[k]) ^= (c[121-k]) ^= (c[k]) ^= (c[121-k]));
```

L'utilisation d'une macro-définition paramétrable *ressemble* à un appel de fonction, mais ce n'est pas un appel de fonction ! Car une macro-définition ne comporte pas un élément essentiel de la déclaration d'une fonction : aucun type n'est défini pour ses paramètres.

Cette caractéristique fait l'intérêt des macro-définitions paramétrables. L'exemple ci-dessus emploie une seule macro-définition pour permuter des variables de n'importe quel type scalaire; si l'on employait des fonctions, on devrait rédiger une fonction distincte pour chaque type de paramètres.

3.3. Mécanismes de substitution avancés

Création d'une chaîne de caractères

Si, dans le texte de remplacement, le nom d'un paramètre est préfixé par le signe **#**, le paramètre effectif est transformé en chaîne de caractères, c'est-à-dire qu'il est placé entre guillemets.

Exemple : macro-définition pour afficher la valeur d'une variable ou d'une expression (peut être employée pour la mise au point des programmes) :

```
#define VOIR(expr,type) printf(#expr " = %"#type "\n",(expr))
       /* expr : expression (variable, etc.)
          type : code %. pour printf() */
```

exemples d'utilisation :
```
int i = 1;
float r = 0.5;
float puiss(float racine, int exp);
.....
      VOIR(i,d);
```

[29] Sinon, la parenthèse ouvrante serait le premier caractère du texte de remplacement d'une macro-définition constante.

```
                        VOIR(puiss(r,2),g);
```

texte transformé :
```
printf("i" " = %""d" "\n",(i));
                    printf("puiss(r,2)" " = %""g" "\n",(puiss(r,2)));
```
texte reçu par le compilateur (les chaînes contiguës sont concaténées) :
```
                    printf("i = %d\n",(i));
                    printf("puiss(r,2) = %g\n",(puiss(r,2)));
```

messages affichés :
```
                        i = 1
                        puiss(r,2) = 0.25
```

Concaténation

Deux parties du texte de remplacement séparées par le double signe **##** sont concaténées, c'est-à-dire collées l'une à l'autre dans le texte résultant. L'intérêt de ce mécanisme réside dans le fait que ces parties de texte peuvent être des paramètres. Cette technique est habituellement employée pour *créer des noms structurés* (de fonctions, de fichiers ...).

Exemple. Un programme comporte notamment l'affichage d'un menu. Ce programme étant destiné à des publics de langues différentes (français, néerlandais...), il existe pour chaque langue une version distincte de la fonction d'affichage du menu. Le programme lui-même sera compilé en différentes versions linguistiques.

```
#define LNG    f  /* choix de la langue de compilation */

   /* prototypes des fonctions menu_? : */
int menu_f(void);       /* menu français */
int menu_n(void);       /* menu néerlandais */

   /* construction du nom de la fonction menu_?() : */
# define MENU(langue)   menu_ ## langue ## ()

.....
oper = MENU(LNG);   /* appel de la fonction d'affichage */
```

texte reçu par le compilateur :
```
oper = menu_f();
```

Redéfinition d'un identificateur

Le texte substitué à l'appel d'une macro-définition est reparcouru pour y chercher des appels d'*autres* macro-définitions connues à ce point du programme; ces appels sont traduits à leur tour. (Restriction : un nom obtenu par une des deux opérations précédentes **#** – chaîne de caractères – ou **##** – concaténation – ne subit plus aucune transformation.) Ce procédé se répète jusqu'à ce que plus aucun appel de macro-définition *non traduit* ne se rencontre dans le texte élaboré.

Exemple :
```
# define DIM  32
# define PERMUTER(a,b)  ((a) ^= (b) ^= (a) ^= (b))
...
... PERMUTER(n[0+i],c[DIM-1-i]);
```

la première transformation donne :
```
((n[0+i]) ^= (n[DIM-1-i]) ^= (n[0+i]) ^= (n[DIM-1-i]));
```
le texte livré par le pré-processeur au compilateur devient :
```
((n[0+i]) ^= (n[32-1-i]) ^= (n[0+i]) ^= (n[32-1-i]));
```

Cependant, si dans le texte substitué à l'appel d'une macro-définition M se retrouve ce *même nom* M, il n'est pas traduit une seconde fois. Grâce à cela, il est possible de redéfinir l'identificateur M.

Exemple. Un programme conçu dans un environnement unilingue sélectionne des messages numérotés et les affiche en appelant une fonction *msg(num)*. Ce programme est revu pour servir dans un environnement polyglotte et une nouvelle fonction *msg()* est créée, qui utilise un deuxième paramètre : un code identifiant la langue dans laquelle doit être affiché le message. Grâce à la macro-définition ci-dessous, les anciens appels, indiquant un seul paramètre, sont automatiquement adaptés pour passer deux paramètres.

```
# define  msg(num)      msg(langue,num)
```

Un appel `msg(13)` sera changé en `msg(langue,13)`.

3.4. Compilation conditionnelle

Soit un texte de la forme :

```
# if macro-expression-1
texte-1
# elif macro-expression-2
texte2
.....
# else
texte-n
# endif
```

Le pré-processeur évalue, dans l'ordre, les macro-expressions;
si l'une d'entre elles est vraie ($\neq 0$), le *texte-i* qui la suit immédiatement est conservé et les autres, détruits;
si aucune n'est vraie et pourvu que figure une ligne `#else`, *texte-n* est conservé et les autres sont détruits.

• Chaque *texte-i* est un texte quelconque. Souvent, il ne contient que des macro-définitions et autres directives # pour le pré-processeur; il peut néanmoins contenir du texte en langage C à proprement parler. Il est possible d'emboîter des constructions `#if` ... `#endif`.

Exemple :

```
#if LANGUE == 'F'
      /* français : "Oui" */
 #define OK  'O'
#elif LANGUE == 'N'
      /* néerlandais : "Ja" */
 #define OK  'J'
#else
      /* anglais : "Yes" */
 #define OK  'Y'
#endif
# define KO  'N'
```

• Chaque macro-expression doit être une expression constante d'un type entier; elle ne peut contenir ni l'opérateur *sizeof* ni aucun opérateur de conversion de *(type)*. Tous les nombres sont interprétés comme étant du type `long int`.

• L'expression **`defined`** ***identificateur*** ou **`defined(`*identificateur*`)`** est vraie (vaut 1), si l'identificateur est déclaré par `#define` . La valeur affectée à cet identificateur n'est pas prise en considération; elle peut donc manquer dans la déclaration.

Exemple : grâce à la définition conditionnelle ci-dessous, l'opération VOIR n'aura d'effet que dans un programme compilé avec une option "TEST"

```
#if defined(TEST) /* avec l'option TEST : afficher un msg */
#define VOIR(expr,type) printf(#expr " = %"#type "\n",(expr))
#else  /* sans l'option TEST : ne rien faire (macro vide) */
#define VOIR(expr,type)
#endif
```

les messages ne sont produits que si, pour la compilation, ce texte est précédé de la ligne suivante :

```
# define TEST
```

Le pré-processeur propose des formes équivalentes :

`#ifdef` *identificateur* ⇔ `#if defined(`*identificateur*`)`
`#ifndef` *identificateur* ⇔ `#if !defined(`*identificateur*`)`

Inclusion conditionnelle

Illustrons une utilisation intéressante de la compilation conditionnelle.

Supposons qu'on ait créé différentes bibliothèques de fonctions mathématiques spécialisées, décrites dans des fichiers d'en-tête : *m_base.h* (fonctions de base), *m_fin.h* (applications financières), *m_stat.h* (applications statistiques), etc. Les fichiers d'application ont besoin des fonctions de base; le fichier *m_base.h* est donc inclus dans chacun des autres :

m_fin.h
```
# include  "m_base.h"
.....
```

m_stat.h
```
# include  "m_base.h"
.....
```

Si les deux fichiers *m_fin.h* et *m_stat.h* sont eux-mêmes inclus dans un même programme, on doit éviter de définir deux fois le contenu de *m_base.h*, ce qui pourrait engendrer des erreurs. Pour cela, la technique suivante est couramment employée : le fichier *m_base.h* définit un nom qui le désigne lui-même et le contenu du fichier n'est compilé que si ce nom n'est pas encore défini :

m_base.h
```
# if !defined(M_BASE)  /* si pas encore inclus, */
 # define M_BASE        /* signaler l'inclusion */
 ........              /*  et traiter le texte */
 ........
# endif
```

3.5. Remarques sur la gestion des macro-définitions

Avec les prototypes de fonctions, les macro-définitions constituent le contenu principal des fichiers d'en-tête *nnnnnn.h* .

L'effet d'une macro-définition s'étend de l'endroit où elle est définie jusqu'à la fin du fichier source. A moins que cet effet soit annulé par une ligne

```
# undef nom
```

Options à la compilation

La commande de compilation comporte une **option `-D`** permettant de définir des macro-constantes, dont la valeur n'est donc pas figée (par `#define`) dans le texte du programme. (On peut ainsi arrêter la dimension d'un tableau au moment de compiler le programme...)

Exemples : fixation de la dimension *N* d'un tableau, par l'option de compilation : `-DN=80`
définition de l'option TEST d'exécution : `-DTEST`

4. *Supplément.* La démonstration des programmes : *assert()*

La proposition *"tout programme peut être obtenu par l'emboîtement de trois types de constructions : la séquence, l'itération, la sélection"* fonde les méthodes dites de **programmation structurée**. Aux yeux de cette méthodologie, un des principaux intérêts de la composition des opérations par emboîtement de constructions est qu'elle rend possible la démonstration qu'un programme est correct :

– une construction (séquence, itération ou sélection) possède *un seul point d'entrée* et *un seul point de sortie*;
– à la condition de ne pas programmer de sauts ... dès lors qu'on est entré dans une construction, *il est certain qu'on en sortira*;

– on peut attacher au point d'entrée de toute construction une *assertion* affirmant la vérité des *pré-conditions* requises pour pouvoir exécuter la construction;
– on peut attacher au point de sortie de toute construction une *assertion* affirmant la vérité des *post-conditions* prouvant que la construction a produit son effet;

– dès que ces deux séries de conditions sont vraies pour une construction, il est certain qu'elles le restent si cette construction est emboîtée dans une autre;
– on peut ainsi dérouler un processus de *démonstration incrémentale* : prouver la correction des constructions les plus intérieures et finir par prouver celle de la construction la plus globale.

Le fichier d'en-tête standard ***assert.h*** contient une macro-définition **`assert(`*`expr de condition`*`)`** permettant de tester les assertions posées dans le programme. Si l'affirmation testée est vraie, l'exécution du programme se poursuit, sinon elle s'arrête après affichage d'un message adéquat.

Exemple : tester une boucle de mise à zéro d'un tableau de dimension *d* :
– à l'entrée dans la boucle, l'indice doit être *0*
– à la fin de la boucle, l'indice doit être *d*
– à l'entrée de chaque pas de la boucle, l'indice doit être compris entre *0* et *d-1*
– à l'issue de chaque pas de la boucle, l'élément courant du tableau doit être à zéro

```
# include <assert.h>

int total_par_mois[12];
int mois;          /* mois 1..12 */
...
mois = 1;                       assert(mois==1);
while (mois<=12)
{                               assert(mois>=1 && mois<=12);
  total_par_mois[mois-1]=0;     assert(total_par_mois[mois-1]==0);
  ++mois;
```

```
}                                   assert(mois==12+1);
```

(il est évident que, dans cet exemple, ces tests ne sont guère utiles ... mais dans un programme plus réaliste !!!)

Lorsque le programme testé est au point, il n'est pas nécessaire d'enlever du texte les appels `assert()`; leur effet est annulé en définissant le macro-identificateur NDEBUG *avant d'inclure le fichier* `<assert.h>` :

```
#define NDEBUG    /* annule l'effet de 'assert()' */
#include <assert.h>
```

`assert()` est une macro-définition conditionnelle dont voici une version simplifiée :

```
# if !defined(NDEBUG)
      /* si le macro-identificateur NDEBUG n'est pas défini :
         --> définition effective du test assert() */
 #define assert(p)   if(!(p)) {                              \
                               printf("Assertion fausse \n"); \
                               exit(1);                       \
                              }
      /* la fonction 'exit(1)' termine l'exécution du programme */
# else
      /* si le macro-identificateur NDEBUG est défini :
         --> annulation (définition vide) du test assert() */
 #define assert(p)
# endif
```

Exercices

Macro-définitions paramétrables

1.

Ecrire une macro-définition *MAX()* qui, ayant pour paramètres deux nombres d'un même type quelconque, renvoie le plus grand des deux. De manière analogue, écrire une macro-définition *MIN()* qui renvoie le plus petit des deux nombres.

2.

Dans un programme conduisant un dialogue au terminal, chaque introduction de données, par appel de la fonction *scanf()*, répond à un message de demande affiché par *printf()*. Ecrire une macro-définition ***pscanf()*** recevant trois paramètres : le texte du message de demande, le format de la réponse, l'adresse de la donnée à lire.

REMARQUES

Le nombre de paramètres d'une macro-définition ne pouvant pas être variable, chaque appel de *pscanf()* peut lire *une seule* donnée.

Puisque le pseudo-appel *pscanf(message,format,&donnee)* possède la forme d'une expression indécomposable, le texte transformé devrait également constituer une expression indécomposable.

Compilation conditionnelle

3.

Adapter la macro-définition ***pscanf()*** pour qu'elle puisse également servir à lire des données dans un fichier – l'affichage du message de demande doit alors être inhibé.

Chapitre 7. Variables et fonctions

Un programme C est un ensemble de fonctions et ces fonctions sont souvent ***réparties dans plusieurs modules*** (fichiers sources) compilés séparément. Se pose dès lors la question de l'**accessibilité** des variables à partir des différentes fonctions, comme celle de l'accessibilité d'une fonction à partir d'une autre.[30]

1. Attributs d'une variable

> **Déclarer** une variable ou une fonction, c'est lui donner un nom *identificateur* en même temps qu'on en précise les *attributs*.

Une **variable** est caractérisée par les attributs suivants :

- son **type**, définissant le domaine où elle prend ses valeurs
  - les types de base sont indiqués par des mots clés du langage (voir chapitre 2);
  - d'autres types, construits au départ des types de base, seront étudiés au chapitre 8;
- la **portée** de son identificateur, qui détermine les endroits du programme d'où elle peut être désignée et atteinte;
- sa **durée de vie** ou sa **classe**, définissant la période pendant laquelle un emplacement de mémoire lui est alloué;
- sa **valeur initiale**;
- le **droit de mise à jour** octroyé aux programmes.

Puisque déclarer une **fonction** revient à déclarer les variables – paramètres et résultat – qui en constituent l'interface, la plupart de ces attributs sont aussi ceux d'une fonction.

1.1. Portée et visibilité

> On appelle **portée** d'un identificateur la partie de texte dans laquelle il peut être utilisé pour référencer l'objet qu'il désigne; on dit également que cet objet est "**visible**" dans cette partie du texte.

Portée locale ou globale d'un identificateur

On peut considérer que le texte d'un programme C est hiérarchisé de la manière suivante :

niveau 0 : programme
  niveau 1 : module – peut contenir la déclaration de variables (globales) et de fonctions
    niveau 2 : fonction – peut contenir la déclaration de variables (paramètres formels)
      niveau 3 : bloc – peut contenir la déclaration de variables (locales)
        niveau 4 : bloc inclus – peut contenir la déclaration de variables (locales)
          .....

- Une **variable locale** est déclarée à l'intérieur d'un bloc (donc aussi d'une fonction); elle ne peut être référencée que par les opérations elles-mêmes situées après sa déclaration à l'intérieur de ce bloc; on dit qu'elle n'est visible que dans ce bloc. ***Dans un bloc, les déclarations doivent précéder les instructions.***

[30] Le présent chapitre complète le chapitre 2.

• Un **paramètre formel** est local à la fonction où il est déclaré; il n'est visible que pour les opérations programmées dans cette fonction.

• Une **variable globale** est définie dans un module, en dehors de toute fonction; elle peut être utilisée par n'importe quelle fonction définie après elle dans le même module. Une variable globale est visible depuis le point où elle est déclarée jusqu'à la fin du texte source.

• En C, une **fonction** est toujours globale. Elle ne peut pas être définie à l'intérieur d'une autre fonction.

Usage privé ou public

• Sauf s'il est déclaré *static*, un objet – variable ou fonction – défini comme global dans un module peut être référencé et utilisé par des fonctions définies dans d'autres modules. On dit qu'un tel objet est **public**.

• Si sa déclaration le qualifie de ***static***, un objet – variable ou fonction – défini comme global est protégé contre tout accès extérieur; il ne peut être atteint que de l'intérieur du module où il se trouve déclaré. Un tel objet est **privé**.[31]

Exemples

La fonction *compter()*, ci-après, compte le nombre de fois où elle est appelée et, chaque fois qu'elle est exécutée, renvoie le numéro courant à la fonction appelante. Pour pouvoir commencer le comptage à une valeur quelconque, une autre fonction *init()* doit initialiser le compteur. Celui-ci doit donc être une variable globale accessible aux deux fonctions. Par essence, les deux fonctions *init()* et *compter()* sont à usage public, tandis que la variable *compteur* est privée et n'est accessible qu'aux fonctions définies dans le même module.

```
#include <stdio.h>

#define PRIVATE static
PRIVATE int compteur = 0;      /* variable globale privée */

void init(void)       /* initialisation facultative du comptage */
{ printf("donnez la valeur initiale : "); scanf("%d",&compteur); }
int compter(void)     /* comptage */
{  return (++compteur);  }  /* le compteur est incrémenté (++)
                               avant d'être renvoyé en résultat */
```

Dans la fonction *puiss()* calculant n^p, une variable locale est nécessaire pour préparer le résultat à renvoyer à la fonction appelante.

```
float puiss(float n,int p)     /* élever n à la puissance p */
{
   float resultat = 1;         /* variable locale */
   for(;p>0;--p) resultat *= n;
   for(;p<0;++p) resultat /= n;
   return (resultat);
}
```

[31] Attaché à une variable locale, le qualificatif ***static*** a une autre signification. Le double emploi, avec double signification, de ce qualificatif unique est malheureux. Suggestion : prendre le détour d'une macro-définition **`#define PRIVATE static`** .

Dans un *programme hiérarchisé*,[32] les variables (données cumulatives, titre, identifiant ...) caractérisant chaque sous-ensemble d'un même niveau de décomposition seront, en principe, déclarées localement à l'intérieur du bloc traitant les sous-ensembles de ce niveau. Nous reproduisons ici l'exemple du chapitre 5.

Lisant des relevés de fabrication par ateliers, on imprime les totaux de fabrication par usines et le total général (il s'agit d'une firme possédant plusieurs usines). Ce programme est hiérarchisé sur trois niveaux : traitement de l'ensemble "firme", traitement de chaque sous-ensemble "usine", traitement de chaque élément "atelier".

```
#include <stdio.h>
int main (void)
{
     /* données lues : */
  short int no_usine;
  short no_atelier;
  int total_atelier;
  int signal;  /* nbre. de données obtenues par scanf()
                  doit être 3 (<0 en fin de fichier) */

     /* 1ère. lecture : */
  signal = scanf("%1hd%2hd %d",&no_usine,&no_atelier,
                               &total_atelier);
  while (signal == 3)
  {    /* traitement d'une firme : */
    long int total_firme;       /* var. locale */
    total_firme = 0;
    while (signal == 3)
    {    /* traitement d'une usine : */
      short int usine_traitee;  /* id. du ss-ens. */
      long int total_usine;     /* var. locale */
      total_usine = 0;
      usine_traitee = no_usine;
      while (signal == 3 && no_usine == usine_traitee)
      {
           /* traitement d'un atelier : */
        total_usine = total_usine + total_atelier ;

           /* lecture suivante : */
        signal = scanf("%1hd%2hd %d",
                       &no_usine,&no_atelier,
                       &total_atelier);
      }
      printf("\nTotal usine %1hd = %ld",usine_traitee,
                                        total_usine);
      total_firme = total_firme + total_usine;
    }
    printf("\nTotal firme = %ld",total_firme);
  }
  exit(0);   /* ou :   return (0);  = terminaison OK */
}
```

[32] Cf. chapitre 5.

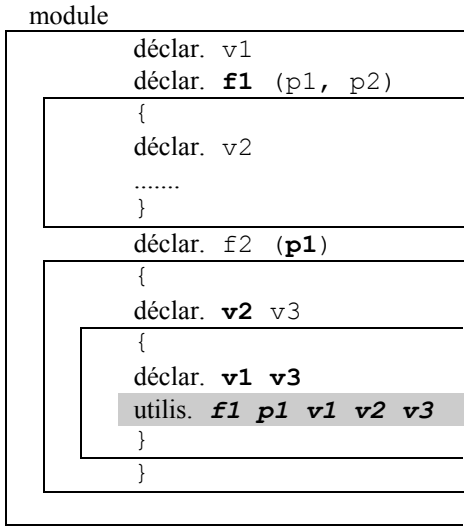
Directive d'utilisation

> On peut poser comme règle générale que toute variable doit être déclarée le plus près possible des instructions qui l'utilisent. ***L'usage et la portée d'un identificateur doivent être les plus petits possibles.***

Règles de visibilité

Soit un identificateur *utilisé* (toujours à l'intérieur d'un bloc, disons de niveau *b*); quelle déclaration s'applique (est visible) à cet endroit du programme ? Le compilateur cherche successivement cette déclaration

dans le bloc de niveau *b*,
dans le bloc de niveau *b − 1* incluant le bloc de niveau *b*,
.....,
dans la fonction (de niveau 2) incluant le bloc de niveau 3,
dans le module (de niveau 1) incluant la fonction.



en **gras** :
déclarations utilisées

• Un même identificateur déclaré dans deux parties disjointes désigne donc deux objets distincts. *Exemples : p1, v2.*

• Un identificateur déclaré dans une partie de programme peut être redéfini dans une partie incluse dans la première (exception : les identificateurs des paramètres formels d'une fonction ne peuvent pas être redéfinis comme variables locales dans un bloc interne à cette fonction). Dans ce cas, la définition la plus intérieure "cache" l'autre, c'est-à-dire que l'objet défini par la première déclaration n'est pas accessible dans la partie incluse. *Exemples : v1, v3.*

Extension de visibilité des déclarations globales (`extern`)

La visibilité d'un objet global, fonction ou variable, peut être étendue au moyen d'un **rappel de définition** externe.

• Les **objets publics** définis dans un premier module sont accessibles aux fonctions définies dans un second module s'ils sont, dans ce second module, redéclarés comme *extern*.

– Le qualificatif ***extern*** doit être explicitement indiqué pour une variable; il est implicite et peut ne pas être écrit pour une fonction.

– Si un rappel de déclaration est rédigé à l'intérieur d'un bloc ou d'une fonction, l'extension de visibilité est limitée à la portée locale; si le rappel de déclaration est rédigé au niveau global du second module, l'extension de visibilité est globale pour ce second module.

– Ces rappels de déclarations sont habituellement rassemblés dans des fichiers d'en-tête *.h* à inclure dans les modules utilisateurs.

module1.c

```
int v_global = 1;
int fonct(void)
{ ..... }
```

module1.h

```
extern int v_global;
    int fonct(void);
```

module2.c

```
#include "module1.h"
```

REMARQUE. C'est le programme relieur qui substitue aux identificateurs des objets publics les adresses d'implantation de ces objets en mémoire.[33] Souvent, le programme relieur est capable de combiner entre eux des modules rédigés dans différents langages (C et COBOL, par exemple); les identificateurs publics doivent alors posséder une forme reconnaissable par ce relieur commun. Les limitations suivantes sont fréquentes : l'identificateur ne peut pas comporter de tiret, sa longueur est limitée (par exemple, à 8 caractères), il n'est pas fait de distinction entre majuscules et minuscules (par exemple, *puiss()* et *PUISS()* constituent un seul identificateur).

• Il est possible de placer dans un module le rappel de déclaration d'une fonction ou d'une variable globale, publique ou privée, définie dans *le même module*. Ce procédé sert surtout pour autoriser l'appel d'une fonction qui ne sera définie que plus loin dans le texte du module. *Exemple :*

pgcdppcm.c

```
int ppcm (int a, int b)
{
   extern pgcd (int a, int b);      /* "rappel" de déclaration */
   return ((a * b) / pgcd (a, b));
}

int pgcd (int a, int b)
{
   if (a == b) return a;
   if (a < b) return pgcd (a, b - a);
   if (a > b) return pgcd (b, a - b);
}
```

1.2. Durée de vie et classe d'une variable

Une variable est *"créée"* et *"détruite"* au cours de l'exécution du programme; on veut dire par là qu'un emplacement de mémoire est *"alloué"* puis *"libéré"* (désalloué), dans lequel les valeurs successives de la variable sont conservées. La **durée de vie** d'une variable est la période s'étendant du moment de sa création à celui de sa destruction. Cette durée de vie est déterminée par la **classe** de la variable.

[33] Pour les autres identificateurs, cette substitution est opérée par le compilateur.

• Une **variable dynamique ou "automatique"** (classe ***auto***) est une variable de portée locale, créée lors de chaque entrée dans le bloc où elle est définie et visible, et détruite lors de chaque sortie de ce bloc; *sa durée de vie est égale à la durée d'exécution du bloc*; ***entre deux exécutions du bloc, sa valeur est perdue***.

• La définition ci-dessus reste valable si l'on remplace *variable locale* par *paramètre formel* et *bloc*, par *fonction*.

• Une **variable statique** (classe ***static***), de portée globale ou locale, est créée au démarrage de l'exécution du programme[34] et détruite à la terminaison du programme; *sa durée de vie est égale à la durée d'exécution du programme*; ***la valeur d'une variable locale statique est conservée entre les exécutions du bloc où elle est définie et visible*** (... mais, puisqu'il s'agit d'une variable locale, elle ne peut être référencée que par les opérations du bloc en question).

Exemple. La fonction suivante compte le nombre de fois où elle est appelée et, chaque fois qu'elle est exécutée, renvoie le numéro courant à la fonction appelante. Si le compteur était (implicitement) déclaré *auto* plutôt que *static*, la fonction renverrait à chaque appel la valeur 1.

```
int compter(void)
{
   static int compteur=0;     /* valeur initiale = 0 */
   return (++compteur);  /* le compteur est d'abord incémenté (++)
                            puis renvoyé en résultat */
}
```

• On peut demander au compilateur de créer une variable "automatique" dans un **registre** du processeur (classe ***register***) plutôt qu'en mémoire centrale. S'il n'y a pas suffisamment de registres disponibles, la classe *register* est automatiquement changée en *auto*. (Le nombre de registres disponibles pour cet usage est, au maximum, de l'ordre d'une dizaine.)[35]
– *Le fait de placer une donnée dans un registre économise les transferts entre mémoire et processeur. Compte tenu de ce que leur nombre est limité, on réservera les registres aux variables auxquelles l'exécution de la fonction accède le plus fréquemment.*

• La classe ***extern*** est celle des **rappels de déclaration** d'une variable globale (et donc statique).

Organisation de la mémoire allouée à un programme

Les classes de variables (que le langage C appelle des "classes de mémoire") reflètent la structure de l'espace de mémoire centrale alloué à un programme C en exécution. Cet espace est logiquement découpé en quatre "segments" (sans compter les registres du processeur) :

- **segments statiques** : le segment des *instructions* et le segment des *données*;
- **segments dynamiques** : la *pile* ("stack") et le *tas* ("heap").

[34] Plus exactement, une variable statique est créée lors du chargement du programme (texte binaire exécutable) en mémoire centrale, c'est-à-dire *avant* que commence l'exécution du programme.
[35] On peut considérer que le résultat d'une expression, en particulier le résultat d'un appel de fonction, est de classe *register*.

Au chargement du programme, c'est-à-dire au démarrage de l'exécution,

- les instructions des différentes fonctions sont créées dans le segment des *instructions*;
- les variables statiques sont créées dans le segment des *données*;
- la *pile* est vide;
- le *tas* est vide.

	instructions	*données*	*pile*	*tas*
		var. globales		
main()	instructions	var. locales statiques		
fonction1()	instructions	var. locales statiques		
fonction2()	instructions	var. locales statiques		

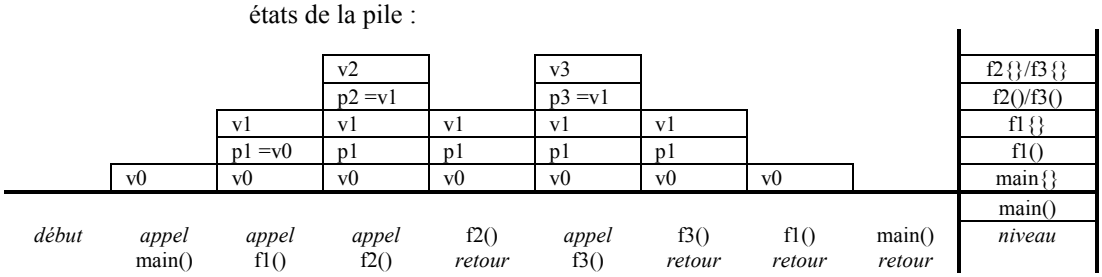
La **pile** ("stack") est la structure dans laquelle sont créés les paramètres formels des fonctions et les variables "automatiques" locales aux différents blocs :

- lorsque l'exécution "entre" dans une fonction ou un bloc, les paramètres ou variables de la fonction ou du bloc sont créés au sommet de la pile;
- lorsque l'exécution "sort" d'une fonction ou d'un bloc, les paramètres ou les variables de la fonction ou du bloc sont enlevés du sommet de la pile.

Exemple : programme schématique : `f` = fonction, `p` = paramètre, `v` = variable

```
f3(p3) {auto v3; ......;}
f2(p2) {auto v2; ......;)
f1(p1) {auto v1; f2(v1);f3(v1);}
main() {auto v0; f1(v0);}
```

états de la pile :



Le mécanisme d'empilement rend possibles les ***appels récursifs*** d'une fonction.

Exemple : empilement des paramètres de la fonction récursive *pgcd(a,b)* – cf. chapitre 1 :

appel n° 4	04 | 04
appel n° 3	08 | 04
appel n° 2	08 | 12
appel n° 1	08 | 20
`main() {pgcd(8,20);}`	

Le **tas** ("heap") est une réserve de mémoire dont le programmeur, en appelant la fonction standard *malloc()*, peut allouer des fragments à différents objets créés dynamiquement par le programme.[36]

1.3. Initialisation d'une variable

La déclaration de certaines variables peut comporter une clause d'**initialisation**. Cette clause détermine quelle valeur "initiale" est rangée dans la variable chaque fois que celle-ci est créée.

A défaut d'une clause d'initialisation explicite,

- une *variable statique*, qu'elle soit locale ou globale, est initialisée à la valeur zéro,
- la valeur initiale d'une *variable automatique* est indéterminée.

Méthode générale

Dans tous les cas, une valeur initiale peut être définie au moyen d'une expression constante, c'est-à-dire ne comportant que des constantes en guise d'opérandes.

Exemples (déclarations et initialisations) :
```
int compteur = 0;
float pi = 3.1416;
float n = 3.0 / 7.0;
int i = 127;
int j = - 1;
char code_langue = 'F';
char nom_fichier[32] = "C:\\exemple.txt";
```

Initialisation des variables agrégats

Dans le cas d'une variable agrégat – un tableau (cf. chapitre 2) ou une structure (cf. chapitre 8) –, une liste d'expressions constantes fournit la valeur de chaque composant; cette liste est placée entre accolades **{ }**.

Exemples :
```
short int nbre_jours_dans_mois [12]
          = {31,29,31,30,31,30,31,31,30,31,30,31};
char jour [7][9] = {"dimanche",
                    "lundi", "mardi", "mercredi",
                    "jeudi", "vendredi", "samedi"};
```

Si un composant est lui-même décomposable, ses valeurs forment une sous-liste elle-même placée entre accolades (la liste d'initialisation ci-dessous est formée de deux sous-listes).[37]

Exemple :
```
short int nbre_jours_dans_mois [2][12]
     /* année bissextile - année ordinaire */
         = { {31,29,31,30,31,30,31,31,30,31,30,31},
             {31,28,31,30,31,30,31,31,30,31,30,31} };
```

[36] Cf. chapitre 9.

[37] Il est toutefois toléré de ne pas hiérarchiser la liste des valeurs, mais ceci ne semble pas recommandable.
Exemple :
```
short int nbre_jours_dans_mois [2][12]
     /* année bissextile - année ordinaire */
     = {31,29,31,30,31,30,31,31,30,31,30,31,
        31,28,31,30,31,30,31,31,30,31,30,31};
```

Si une liste de valeurs initiales est incomplète, les valeurs manquantes sont automatiquement remplacées par zéro.

Initialisation des variables automatiques

Une variable automatique est ***réinitialisée*** chaque fois qu'elle est créée, c'est-à-dire chaque fois que l'on commence une exécution du bloc où elle est déclarée.

La valeur initiale d'une variable automatique peut être définie par la méthode générale des expressions constantes. Elle peut également être définie par une ***expression variable***, c'est-à-dire comportant des variables ou des appels de fonctions. Les variables ou fonctions référencées doivent évidemment être déjà déclarées et visibles.

Exemples :
```
int i = 0;                      /* init. par constante */
int j = i+1;                    /* init. par variable */
int no_facture = compter();     /* init. par fonction */
```

Contrairement à ce qui se passe pour une initialisation par des expressions constantes, *une seule* expression variable est autorisée. En d'autres termes, l'initialisation par expressions variables n'est pas applicable aux tableaux.

> Pourquoi ce mode d'initialisation est-il impossible pour les variables statiques ? Une *variable statique* est créée pendant le chargement du programme en mémoire centrale, sa valeur initiale est donc établie *avant l'exécution* du programme; en réalité, cette valeur est calculée par le compilateur, qui l'inscrit dans le texte binaire à charger en mémoire centrale. En revanche, une *variable automatique* est créée *pendant l'exécution* du programme; sa création peut donc avoir recours à une fonction du programme ou copier le contenu d'une autre variable du programme.

Exemple général

Dans un *programme hiérarchisé*, les variables (données cumulatives, titre, identifiant ...) caractérisant chaque sous-ensemble d'un même niveau de décomposition sont, en principe, déclarées localement à l'intérieur du bloc traitant les sous-ensembles de ce niveau; leur déclaration peut les initialiser. Nous reproduisons ici, en le modifiant légèrement, l'exemple repris plus haut.

```
#include <stdio.h>
int main (void)
{
     /* données lues : */
  short int no_usine;
  short int no_atelier;
  int total_atelier;

     /* 1ère. lecture : */
  int signal = scanf("%1hd%2hd %d",
                     &no_usine,&no_atelier,
                     &total_atelier);
     /* le nbre. de données obtenues doit être 3 */
  while (signal == 3)
  {    /* traitement d'une firme : */
    long int total_firme = 0;   /* var. locale */
    while (signal == 3)
    {    /* traitement d'une usine : */
      short int usine_traitee = no_usine;
           /* id. du ss-ens. initialisé par variable */
      long int total_usine = 0;       /* var. locale */
      while (signal == 3 && no_usine == usine_traitee)
      {
           /* traitement d'un atelier : */
        total_usine = total_usine + total_atelier ;

           /* lecture suivante : */
        signal = scanf("%1hd%2hd %d",
                       &no_usine,&no_atelier,
                       &total_atelier);
      }
      printf("\nTotal usine %1hd = %ld",usine_traitee,
                                          total_usine);
      total_firme = total_firme + total_usine;
    }
    printf("\nTotal firme = %ld",total_firme);
  }
  exit(0);   /* ou :   return (0);  = terminaison OK */
}
```

1.4. Droit de mise à jour d'une variable

Une variable ou un paramètre déclaré dans un programme peut être mis à jour (sa valeur peut être modifiée) par toute opération de la partie de ce programme où la variable ou le paramètre en question est visible.

Il est possible de modifier ce droit. Une variable ou un paramètre peut être déclaré :

- **volatile** : la donnée peut être mise à jour par un élément extérieur au programme
  – ceci permet, par exemple, de définir une variable "tampon" pour l'échange de données entre deux programmes s'exécutant simultanément;

• **const** ("constant") : la donnée ne peut pas être mise à jour par les opérations du programme
– pour que la déclaration ait un sens, elle doit comporter une clause d'initialisation;[38]
– on définit ainsi une constante *nommée* (dans le texte du programme, ce nom sera plus parlant que la valeur codée qu'il représente) et dont le nom peut avoir une *portée* locale;

exemples :
```
const short int vrai = 1;
const short int faux = 0;
const float pi = 3.1416;
const char homme = 'M';
const char femme = 'F';
```

• une même donnée peut être déclarée à la fois *const* et *volatile*.

Note. Les mots *const* et *volatile* sont appelés des qualificatifs (du nom) de type.

REMARQUE. Pour économiser la réinitialisation d'une *variable constante locale* à chaque entrée dans le bloc auquel elle appartient, il est intéressant de la déclarer de classe `static` . Ceci est particulièrement indiqué dans le cas d'un *tableau*.

Exemple :
```
const static short int nbre_jours_dans_mois[12]
     = {31,29,31,30,31,30,31,31,30,31,30,31};
```

2. Déclaration des variables et fonctions

2.1. Canevas général

Le canevas général d'une déclaration de variable est le suivant :

droit *classe* ***type*** ***identificateur*** *initialisation* **;**

Une clause d'***initialisation*** prend une des deux formes suivantes :

pour toute variable :	pour une variable *auto* seulement :
= **{** *liste d'expressions constantes* **}**	**=** *expression variable*

Pour une variable scalaire, une seule expression est utilisée et les accolades **{}** peuvent être omises.

Les tableaux ci-dessous indiquent quels éléments de ce canevas sont autorisés dans chaque cas.

– ***identificateur*** : la déclaration d'un identificateur est toujours obligatoire. – ***type*** : la déclaration d'un type de données est obligatoire, sauf dans la déclaration d'une fonction.

Remarque. Si, dans une déclaration, le mot **`int`** est précédé d'au moins un autre mot, il peut être omis.

[38] Exception possible : une variable *volatile*.

déclaration primaire	*portée (visibilité)*	*classe implicite*	*classes explicites*	*type implicite*	*initialisation autorisée*[c]	*droits de mise à jour*[e]
variable globale	globale (module)	static[b] (public)	static[b] (privé)		oui	const volatile
fonction[a]	globale (module)	static[b] (public)	static[b] (privé)	int	non	[f]
paramètre formel	locale (fonction)	auto	register		non[d]	const volatile
variable locale	locale (bloc)	auto	auto register static		oui	const volatile

a La déclaration d'une fonction équivaut à la définition d'une variable résultat de cette fonction.
b Un objet global déclaré explicitement *static* est privé; sinon il est public.
(cet emploi du qualificatif *static* est malheureux).
c Une variable *static*, si elle n'est pas explicitement initialisée, l'est implicitement à la valeur 0.
d L'initialisation des paramètres formels est effectuée par l'appel de la fonction.
e Les deux qualificatifs *const* et *volatile* peuvent figurer dans la même déclaration.
f Les qualificatifs *const* et *volatile* sont autorisés dans la déclaration d'une fonction, bien qu'ils n'aient dans ce cas aucune utilité.

rappel de déclaration[a]	*portée (visibilité)*	*classe implicite*	*classe explicite*	*type implicite*	*initialisation autorisée*	*droits de mise à jour*[c]
variable globale			extern[b]		non	const volatile
fonction		extern	extern	int	non	[d]

a L'objet ainsi redéfini doit être un objet global.
Cet objet doit être public (non *static*) si sa définition primaire figure dans un autre module.
b Le qualificatif *extern* est obligatoire.
c Les deux qualificatifs *const* et *volatile* peuvent figurer dans la même déclaration.
d Les qualificatifs *const* et *volatile* sont autorisés dans la déclaration d'une fonction, bien qu'ils n'aient dans ce cas aucune utilité.

2.2. Déclaration des variables

Déclaration primaire

– Les **variables globales** sont déclarées dans un module, en dehors de toute fonction, mais avant la définition des fonctions qui doivent y avoir accès. Habituellement, toutes les variables globales sont définies en tête du module.
– Les **variables locales** sont déclarées au début d'un bloc à l'intérieur d'une fonction, avant toute opération interne à ce bloc.

On peut, en une seule déclaration, définir plusieurs variables possédant des attributs identiques, à l'exception de la valeur initiale. Une telle déclaration comporte plusieurs couples *identificateur initialisation* :

droit	*classe*	***type***	***identificateur***	*initialisation*	**,**
			identificateur	*initialisation*	**,**
			.....		**;**

La mention du *type* et l'*identificateur* sont obligatoires.

Exemples :
```
unsigned int prix_unitaire;
float surface, volume;
register short i;
signed long int total_du_mois [12];
char nom_fichier[32] = "C:\exemple.txt";
const char espace = ' ', tab = '\t', fin_ligne = '\n';
const short int vrai = 1, faux = 0;
const float ponder = 4.0 / 9.0;
const static short nbre_jours_dans_mois[12]
                  = {31,29,31,30,31,30,31,31,30,31,30,31};
   /* étant initialisées par des expressions variables,
      les déclarations suivantes doivent être locales : */
int i = 0, j = i+1;
unsigned no_facture = compter();    /* init. par fonction */
```

Rappel de déclaration des variables globales

Le rappel de déclaration des variables globales utilise le même format qu'une déclaration primaire, avec les adaptations suivantes :

- le qualificatif de classe ***extern*** est obligatoire;
- la clause d'*initialisation* n'a aucun sens et elle est interdite.

Exemples :
```
extern unsigned compteur;
const extern short vrai,faux;
```

– Un rappel rédigé en dehors de toute fonction a une **portée globale**.
– Un rappel rédigé au début d'un bloc, à l'intérieur d'une fonction, a une **portée locale**.

2.3. Déclaration des fonctions (format ANSI)

Déclaration primaire

La déclaration primaire d'une fonction comporte, dans l'ordre les éléments suivants :

droit	*classe*	*type*	***identificateur***	**(** *déclaration des paramètres formels*	**)**

L'*identificateur* et les deux parenthèses **( )** sont obligatoires.
Cette déclaration est suivie du bloc **{ }** définissant le corps de la fonction.

Exemples :
```
float puiss (float racine, register int exposant)
int pgcd(int a,int b)
short test_date(const long int date)
```

• **droit de mise à jour** : bien qu'ils n'aient dans ce contexte aucune signification, la déclaration d'une fonction peut comporter les qualificatifs ***const*** et ***volatile***.

• **classe** : la classe indique l'usage privé ou public de la fonction.

- une fonction déclarée sans qualificatif de classe est publique;
- le seul qualificatif autorisé est ***static***; il rend la fonction privée.

• **type** : le type mentionné est celui de la valeur résultat de la fonction.

- le qualificatif ***int*** (entier) peut être laissé implicite;
- le pseudo-type ***void*** ("vide") doit être indiqué pour une fonction qui ne rend pas de résultat.

• **liste des paramètres** :

- la liste comporte la *déclaration* complète de chaque paramètre;
- les parenthèses **( )** sont obligatoires, même si la fonction ne reçoit aucun paramètre;
- l'absence de paramètre est indiquée par le nom du pseudo-type ***void*** ("vide").

Exemples : `int compter(void)` *ou* `compter(void)`
`void init(void)`

Déclaration des paramètres formels

La déclaration d'un paramètre prend la forme suivante :

droit *classe* ***type*** ***identificateur***

La mention du *type* et l'*identificateur* sont obligatoires.
Les déclarations successives sont séparées par des *virgules*.

• **doit de mise à jour** :

- un paramètre peut être déclaré ***const***, c'est-à-dire non modifiable.

• **classe** :

- par défaut, un paramètre formel est de classe *auto*;
- le seul qualificatif de classe autorisé est ***register***.

• **type** :

- le pseudo-type ***void*** ("vide") est employé pour signaler l'absence de paramètres; bien que cela n'ait aucune utilité, le qualificatif *void* peut être suivi d'un identificateur.

• **initialisation** :

- à chaque appel d'une fonction, chaque paramètre formel reçoit comme valeur initiale (copie de) la valeur du paramètre effectif correspondant;
- dès lors, une déclaration de paramètre ne comporte jamais de clause d'initialisation.

Rappel de déclaration : prototype

Le prototype (rappel de signature) d'une fonction reproduit les éléments suivants de sa déclaration primaire :

droit *classe* *type* ***identificateur*** **(** *déclaration des paramètres formels* **)** **;**

Le prototype est suivi du terminateur `;`

• **classe** : le seul qualificatif de classe autorisé est le qualificatif ***extern***; il peut être laissé implicite.

• **paramètres** : dans le prototype d'une fonction, la déclaration de chaque paramètre peut être réduite à la seule indication de son **type**; les autres éléments de la déclaration ont une utilité purement documentaire.[39] Soulignons que la mention d'un identificateur "bien choisi" améliore grandement la lisibilité du programme.

Exemples :
```
int pgcd(int,int);        ou     extern int pgcd(int,int);
int ppcm (int, int);      ou     extern int ppcm (int, int);
float puiss(float,int);    ou
float puiss (float racine, int exposant);
short test_date(const long int date);
int compter (void);
```

• **portée** :

un rappel rédigé en dehors de toute fonction, a une portée globale;
un rappel rédigé au début d'un bloc, à l'intérieur d'une fonction, a une portée locale.

Comme pour les variables, on peut déclarer en une seule rubrique plusieurs fonctions possédant des attributs identiques, à l'exception de la liste de paramètres.

droit *classe* *type* ***identificateur*** **(** *déclaration des paramètres formels* **)** **,** ***identificateur*** **(** *déclaration des paramètres formels* **)** **,** ..... **;**

Exemple :
```
int pgcd(int,int), ppcm(int,int);
```

2.4. Déclaration des fonctions (ancienne forme)[40]

Déclaration primaire

La déclaration primaire d'une fonction comporte, dans l'ordre les éléments suivants :

classe *type* ***identificateur*** **(** *liste des paramètres formels* **)**

Cette déclaration est suivie de celle des paramètres puis du bloc **{ }** définissant le corps de la fonction.
Les paramètres sont définis à la manière des variables.

---

[39] L'autorisation d'écrire, par exemple, `(int,int)` pour signifier la présence de deux paramètres justifie l'obligation de répéter le nom de type pour chaque paramètre, contrairement à ce qui est permis pour les variables.

[40] A titre transitoire, cette ancienne forme de déclaration d'une fonction est conservée dans la norme ANSI pour permettre d'encore compiler d'anciens programmes. Il est recommandé de ne plus l'employer.

Exemples :
```
float puiss (racine, exposant)
   float racine;
   register int exposant;
  { ........ }
int pgcd (a,b)
   int a,b;
  { ........ }
```

- **liste des paramètres** :

  – la liste énumère les ***identificateurs*** des paramètres formels, séparés par des *virgules*;
  – les parenthèses **( )** sont obligatoires, même si la fonction ne reçoit aucun paramètre.

Rappel de déclaration

Le rappel de déclaration d'une fonction reproduit les éléments suivants de sa déclaration primaire :

```
classe  type  identificateur  ()  ,
              identificateur  ()  ,
              .....                ;
```

Le rappel de déclaration est suivi du terminateur `;`

Le prototype d'une fonction ne mentionne pas ses paramètres; néanmoins, les parenthèses **( )** sont obligatoires – elles servent à distinguer la déclaration d'une fonction de celle d'une variable.

Exemples :
```
float puiss ();          ou    extern float puiss();
int pgcd(), ppcm();      ou    extern int pgcd(), ppcm();
```

Exercices

Rédiger et compiler une fonction ne contenant pas d'instructions, mais seulement

- toutes sortes de déclarations de variables et de tableaux,
- des prototypes de fonctions (en format ANSI).

Chapitre 8. Les types construits

1. Méthodes de construction de types de données

1.1. Généralités

Rappel : le **type** d'une donnée est l'attribut de cette donnée qui définit à la fois le ***domaine*** où elle prend ses valeurs et le mode de ***représentation*** de ces valeurs. La définition complète d'un type de données comporte également la spécification précise des ***opérations*** possibles sur les données de ce type – que l'on songe, par exemple, à l'opération `%`, qui ne s'applique qu'aux types entiers, mais aussi aux différences de comportement de la division entière et de la division réelle.

A l'instar des autres langages de programmation, le langage C propose au programmeur des *types prédéfinis*, identifiés par des noms réservés (`int`, `float`, etc.) et propose certaines *méthodes pour définir des types* particuliers, adaptés aux problèmes à traiter. Ces méthodes sont l'***énumération***, la ***structure*** et l'***union***. Toutes ces méthodes "construisent" de nouveaux types de données au moyen d'une liste de composants, généralement appelés *membres*.[41] Voici le schéma de base commun de déclaration d'un type construit :

méthode_de_construction étiquette_de_type **{** *liste des membres* **}** **;**

Exemples :
```
enum code_sexe {SEXE_INCONNU = ' ',
                MASCULIN = 'M', FEMININ = 'F'};
     /* 3 valeurs permises */
struct date {short int jour, mois, annee;};
     /* combinaison de 3 entiers */
union nombre {long int entier; float reel;};
     /* 2 interprétations possibles : entier OU réel */
```

enum, struct, union sont les noms prédéfinis des méthodes de construction
code_sexe, date, nombre sont les étiquettes que le programmeur attribue aux types définis
entre accolades *{ }* sont listés les membres composants du type construit

Après avoir ainsi déclaré un type, on peut déclarer des variables ou des fonctions appartenant à ce type.

Exemples :
```
enum code_sexe  sexe_enfant;
struct date  date_echeance, date_paiement;
struct date  date_du_jour(void);
union nombre  resultat;
```

enum code_sexe, struct date, union nombre identifient les types préalablement construits;
sexe_enfant, date_echeance, date_paiement, resultat sont des identificateurs de variables;
date_du_jour est l'identificateur d'une fonction sans paramètres renvoyant la date du jour en cours.

[41] La **programmation par objets** – qui, au lieu de type de données, parle de *classe d'objets* – permet également de construire les opérations applicables aux objets d'une classe. Elle appelle *méthodes* ces opérations.

Une écriture simplifiée permet de définir en une seule déclaration et le type et les variables qui le possèdent.

Exemples :
```
enum code_sexe {SEXE_INCONNU = ' ',
                MASCULIN = 'M', FEMININ = 'F'}
    sexe_enfant;
struct date {short int jour, mois, annee;}
      date_echeance, date_paiement;
union nombre {long int entier; float reel;}
     resultat;
```

Dans ces déclarations simplifiées, l'étiquette de type (en *italique* dans les exemples ci-dessus) est facultative; on l'omet donc le plus souvent.[42]

Exemples :
```
enum {SEXE_INCONNU = ' ',
      MASCULIN = 'M', FEMININ = 'F'}  sexe_enfant;
struct {short int jour, mois, annee;}
      date_echeance, date_paiement;
union {long int entier; float reel;}  resultat;
```

Portée des déclarations de types

Comme pour la déclaration d'une variable, l'endroit du texte où est rédigée la déclaration d'un type définit la portée, globale ou locale, de cette déclaration.

La déclaration d'un type doit être visible à tout endroit du texte où elle est référencée, en particulier à tout endroit où on déclare une variable ou une fonction de ce type.

> Une variable ne sert qu'à mémoriser une information entre l'exécution de deux instructions, elle existe dans le contexte d'utilisation formé de ces instructions; autant que possible, on déclare donc les variables localement. En revanche, il est dans la nature de la plupart des déclarations de types d'avoir une **portée globale**, et même de figurer dans les fichiers d'en-tête *.h* .

REMARQUE

Une étiquette de type n'est jamais employée seule; elle est toujours précédée du nom de la méthode de construction; il n'y a donc aucun inconvénient technique à employer le même nom comme étiquette de type et comme identificateur de variable : pour le compilateur, la confusion n'est pas possible. Ceci peut néanmoins entraîner des difficultés de lecture pour le programmeur; on préférera donc l'éviter.[43]

Exemples :
```
enum code_sexe  code_sexe;
struct date  date;
```

On n'a pas étendu cette faculté jusqu'à autoriser, par exemple, qu'une étiquette de structure soit la même qu'une étiquette d'énumération. ***Toutes les étiquettes de types visibles à un endroit du programme doivent être distinctes.***

---

[42] Si l'on déclare une étiquette de type, celle-ci peut être employée dans des déclarations ultérieures de variables ou de fonctions. Exemple:
```
struct date {short int jour, mois, annee;}
                        date_echeance, date_paiement;
struct date  date_du_jour(void);
```

[43] Une autre raison d'éviter cette manière de faire tient en ceci. Parce que le langage C++ est un sur-ensemble du langage C, un programme C peut être traité par un compilateur C++. Or, *en C++, une étiquette de type est un nom de type*, que l'on peut employer seul et qui ne peut dès lors désigner en même temps une variable. Sur ce point, il y a incompatibilité entre les deux langages.

NOTE. Référence à un type construit défini ultérieurement

Dans une déclaration *qui ne doit pas connaître la taille de l'espace de mémoire à allouer*, il est permis de référencer un type construit qui ne sera défini (par sa liste de composition) que dans une déclaration ultérieure.

Exemple : déclaration d'une fonction calculant une date d'échéance
la déclaration du type du résultat anticipe sur la déclaration du type d'un paramètre

```
struct date echeance
      (struct date{short jour,mois,annee;} date_depart,short delai)
{  ..............  }
```

1.2. Types codifiés : les énumérations

Un type énuméré est un sous-type du type `int`. Le mode de *représentation* en mémoire est le même et les *opérations* applicables sont les mêmes. La différence réside en ce que le *domaine* (ensemble des valeurs permises) est restreint aux valeurs énumérées dans la déclaration.

Exemple :
```
enum code_sexe {SEXE_INCONNU = ' ',
                MASCULIN = 'M', FEMININ = 'F'};
      /* 3 valeurs permises :  ' '   'M'   'F'  */
```

De plus, chaque valeur (constante) de ce type est identifiée par le nom qui lui est attaché. Ce nom peut être employé en lieu et place de la valeur constante identifiée.

Exemple : si on a fait la déclaration de l'alinea précédent,
on peut programmer `if (sexe_enfant == FEMININ)`
au lieu de programmer `if (sexe_enfant == 'F')`

Déclaration

`enum` *étiquette_de_type* `{` *liste des membres* `}` `;`

Chaque membre d'une énumération est une valeur constante, à laquelle est attaché un nom qui l'identifie.

Différentes variantes existent dans la manière de définir et nommer chaque valeur membre du domaine.

• Identification explicite : chaque nom de valeur se voit correspondre explicitement la valeur identifiée :

```
enum code_sexe {SEXE_INCONNU = ' ', MASCULIN = 'M', FEMININ = 'F'};
```
bien que les constantes soient indiquées en format `char`*, leur représentation est celle d'un* `int`

• Identification implicite : la liste ne mentionne que les noms de valeurs; ces noms désignent, dans l'ordre, les valeurs entières 0, 1, 2, etc. :

```
enum valeur_logique {FAUX, VRAI};
    ⇔   enum valeur_logique {FAUX = 0, VRAI = 1};
```

• Méthode mixte : la liste n'indique que certaines valeurs; les valeurs manquantes se déduisent par incrémentation des précédentes :

```
enum jour {DIMANCHE=1,LUNDI,MARDI,MERCREDI,JEUDI,VENDREDI,SAMEDI};
          /* jours de 1 à 7 */
```

Il est permis d'identifier la même valeur par différents noms. Exemples :

```
enum code_sexe {SEXE_INCONNU = ' ',
                MASCULIN = 'M', FEMININ = 'F',
                HOMME = 'M', FEMME = 'F'};
enum valeur_logique {FAUX, VRAI, NON = 0, OUI};
    ⇔   enum valeur_logique {FAUX=0, VRAI=1, NON=0, OUI=1};
```

Si la même valeur peut être désignée par différents noms, il est évident que chaque nom figurant dans une déclaration doit être univoque; en d'autres termes, tous les noms apparaissant dans une même déclaration – étiquette du type ou noms de membres – doivent être distincts.

Utilisation

• ***Variables***

Il est permis de déclarer (et initialiser) des **variables** appartenant à un type énuméré.

Exemples : étant données les déclarations ci-dessus, on peut écrire :

```
enum code_sexe  sexe_enfant;
enum valeur_logique  option = NON, reponse;
enum jour  aujourdhui;
```

Ni à la compilation du texte, ni à l'exécution du programme ne sont mis en oeuvre des mécanismes qui vérifieraient qu'à tout moment la valeur d'une variable ainsi déclarée est une des valeurs énumérées dans la déclaration de son type.[44] *L'intérêt de telles déclarations est donc purement documentaire : dire qu'une variable est du type "valeur_logique" ou du type "code_sexe" est plus parlant que de la déclarer simplement du type* `int`.

• ***Fonctions***

On peut déclarer des fonctions dont le résultat appartient à un type énuméré.

Exemple :
```
enum valeur_logique  test_date (struct date);
          /* test de la validité syntaxique d'une date */
```

• ***Constantes***

Toute valeur explicitement ou implicitement mentionnée dans une énumération est une **constante** appartenant au type déclaré. Cette constante peut être désignée par le nom qui lui est attaché.

[44] A l'exécution, il faudrait, après toute modification de la variable, comparer sa valeur à chacune des valeurs autorisées par son type. Ceci serait très pénalisant par rapport au temps d'exécution.

Exemples : étant données les déclarations ci-dessus, on peut programmer :

```
if (sexe_enfant == FEMININ)  printf ("fille");
option = OUI;
resultat = (VRAI - resultat); /* inversion logique :
                                    0 -> 1   1 -> 0   */
if (aujourdhui > DIMANCHE)  printf ("jour ouvrable");
```

Portée des déclarations

Lorsqu'on désigne une constante par son nom, cette désignation doit être non ambiguë. Un nom de constante ne peut donc apparaître qu'une seule fois parmi l'ensemble des noms définis par toutes les déclarations visibles à un endroit du programme : déclarations `enum` et déclarations de variables ou fonctions.

Si l'on respecte la directive générale de rendre globales toutes les déclarations de types, tous les noms identificateurs de constantes énumérées doivent donc être univoques. Pour les distinguer des identificateurs de variables ou de fonctions, on les écrit habituellement en majuscules.

Remarque méthodologique

Le langage C offre plusieurs techniques pour indiquer une constante. Quelle forme choisir ?

• On emploiera une **constante littérale**, numérique ou alphabétique, chaque fois que cette constante joue son *rôle de valeur numérique ou de valeur alphabétique*.

Exemples : initialisation d'un compteur : `p = 0;`
test d'appartenance à l'alphabet : `if (caract>='A' && caract<='Z') ...`

• On emploiera un nom de constante énumérée chaque fois qu'une valeur numérique ou alphabétique est employée pour *représenter un concept* d'une autre nature. Le rôle méthodologique d'une déclaration **`enum`** est celui d'une ***table de codification***.

Exemples :
```
enum valeur_logique {FAUX, VRAI};
enum code_sexe {SEXE_INCONNU=' ',HOMME='M',FEMME='F'};
.....
resultat = FAUX;
if (sexe_enfant == SEXE_INCONNU) ...
```

• On emploiera une macro-définition **`#define`** pour une constante dont la valeur *dépend de l'environnement* pour lequel le programme est créé.

Exemple :
```
   /* extrait d'un compilateur : */
#if defined C           /* pour le langage C */
#define LONG_MOT  31
#define TIRET     '_'
#endif
#if defined COBOL       /* pour le langage COBOL */
#define LONG_MOT  30
#define TIRET     '-'
#endif
.....
char mot[LONG_MOT+1];
```

1.3. Types composites : les structures

Pour transmettre une information utile, les données manipulées par un programme doivent être associées dans des relations où chacune joue un rôle déterminé et connu. Exemples :

- un triplet de nombres entiers jouant les rôles de {numéro de jour, numéro de mois, millésime} compose une date;
- un autre triplet d'entiers jouant les rôles de {numéro de code, quantité en stock, stock plancher} renseigne sur la disponibilité d'un article en magasin;
- un couple de chaînes de caractères jouant les rôles de patronyme et de prénom identifie une personne;
- .....

Ces relations peuvent être établies dynamiquement par les opérations du programme. Une structure est un moyen de matérialiser – en mémoire centrale et dans les fichiers – une telle relation, en figeant les rôles.

Une variable de type structuré est une zone de mémoire découpée en *champs*, dont chacun contient une variable *membre*, d'un type déterminé.

Exemple : *date*

27	11	1993
jour	*mois*	*annee*

Déclaration

`struct` *étiquette_de_type* `{` *liste des membres* `} ;`

Chaque membre d'une structure est une variable interne à cette structure. Comme le montrent les exemples ci-dessous, la déclaration des membres prend donc la forme des déclarations de variables.

Exemple :

– *déclar. de type :*
```
struct date {short int jour;
             short int mois;
             short int annee;};
```
ou :
```
struct date {short int jour, mois, annee;};
```

– *déclar. de variables :*
```
struct date  date_echeance, date_paiement;
```

– *représentation en mémoire :*

jour	*mois*	*annee*

Les rôles des variables membres sont identifiés par leurs noms. Dans la représentation matérielle, les différents membres sont contenus dans des champs qui se suivent dans l'ordre de déclaration des membres.

Les membres d'une structure peuvent être de types quelconques, scalaires ou non.

Exemples.

La structure *fiche_produit* a notamment pour membre un tableau *nom_produit* :

– *déclar. de type :*
```
struct fiche_produit
      { unsigned int no_produit;
        char nom_produit[12+1];
        int prix_de_vente;
        float taux_tva;          };
```

Une structure (*entete_commande*) peut avoir pour membre une structure d'un autre type (*date*) :

– *déclar. de types :*
```
/* définition du type 'bon de commande'
   comprenant des 'lignes' de 2 types
   - le type de ligne est indiqué par un code */

  /* ligne d'en-tête : */
struct entete_commande
      { char code_type_ligne;   /* 'C' */
        unsigned int no_commande,
                     no_client;
        struct date date_commande; };

  /* ligne de corps : */
struct ligne_commande
      { char code_type_ligne;   /* 'L' */
        unsigned int no_commande,
                     no_produit,
                     qte_commande; };
```

On peut définir un tableau de structures ...

– *déclar. de type :*
```
  /* bon de commande: en-tête + 15 lignes max. */
struct bon_commande
      { struct entete_commande  entete;
        struct ligne_commande  corps [15]; };
```

– *déclar. de variables :*
```
struct bon_commande  commande_en_cours,
                     commande_archivee;
```

• ***Déclaration des membres***

La déclaration des membres d'un type structuré possède la forme des déclarations de variables. Cependant, compte tenu que cette déclaration prend place dans le cadre d'une définition de type (c'est-à-dire la description d'un modèle), et non pas d'une définition de variable effective (c'est-à-dire la réservation d'un espace en mémoire centrale), la déclaration des membres ne comporte ni l'indication d'une classe de mémoire ni une clause d'initialisation.

> Outre les mentions obligatoires de toute déclaration de variable – type[45] et identificateur –, on peut expliciter pour chaque membre les droits d'accès *const* et *volatile*.

[45] Le type d'un membre ne peut pas être un type défini ultérieurement (cf. supra).

On pourrait donc faire, par exemple, les déclarations suivantes :

```
struct entete_commande
      {const char code_type_ligne;   /* 'C' */
       unsigned int no_commande, no_client;
       struct date date_commande; };

struct ligne_commande
      {const char code_type_ligne;   /* 'L' */
       unsigned int no_commande, no_produit, qte_commande;};
```

Dans ce cas, il sera impossible pour les programmes de modifier le contenu des champs *code_type_ligne*, dont la valeur ne pourra être définie que par une clause d'initialisation à la déclaration des variables.

• ***Définition mathématique : structure = relation***

Comme tout type de données, un type structuré définit un ensemble ou domaine de valeurs. D'un point de vue mathématique, **si nous disons que le type de chaque membre d'une structure représente un sous-domaine de valeurs, le *type structuré* constitue une relation entre ces sous-domaines**, c'est-à-dire un sous-ensemble de leur produit cartésien. Chaque combinaison de valeurs admissible – c'est-à-dire toute valeur d'une *variable structurée* – est un n-uplet de cette relation.[46]

Soit la déclaration `struct date { short int jour, mois, annee; };`

Si l'on se restreint aux seules valeurs admissibles,
- le domaine *jour* contient 31 valeurs (1 à 31),
- le domaine *mois* contient 12 valeurs (1 à 12),
- le domaine *annee*, codé sur 4 chiffres, contient 10.000 valeurs (0000 à 9999);

– leur produit cartésien contient 31 x 12 x 10.000 = 3.720.000 n-uplets.

Certains mois ayant moins de 31 jours, il est clair que le domaine *date* contient moins de valeurs et ne représente donc qu'une partie de ce produit cartésien.

Opérations (1). Taille d'une structure

L'opérateur ***sizeof*** peut être utilisé pour connaître la taille d'une structure. On peut lui désigner comme opérande soit un type structuré, soit une variable structurée.

Exemples :
```
sizeof (struct bon_commande)  /* taille d'un type */
sizeof commande_en_cours      /* taille d'une variable */
```

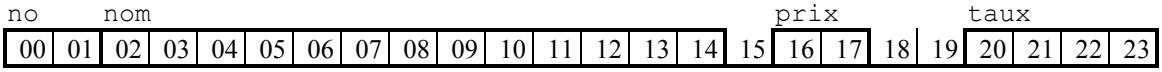
• ***Alignement des membres en mémoire***

Soit *t* la taille (*sizeof*) en octets d'un type scalaire. Sur certaines machines, toute variable scalaire est "alignée" en mémoire, c'est-à-dire placée à une adresse multiple de sa taille *t*. Dans ce cas, il se peut que certains membres d'une structure n'occupent pas des positions contiguës, mais soient séparés par des positions "de remplissage" au contenu indéterminé.

[46] Cette définition mathématique constitue la proposition de départ du *modèle relationnel* des données de CODD, modèle sur lequel reposent les *systèmes de bases de données relationnels*.

Exemple. Si la taille d'une variable `int` est 2 et si la taille d'une variable `float` est 4, la représentation d'une *fiche_produit* dans une mémoire où les données doivent être "alignées" comportera deux zones de remplissage.

```
struct fiche_produit { unsigned int no_produit;
                       char nom_produit[12+1];
                       int prix_de_vente; float taux_tva; };
```



Dans une telle situation, la taille de la structure est supérieure à la somme des tailles de ses membres.

Opérations (2). Désignation des membres

Tout programme peut agir sur les membres d'une variable structurée. La désignation d'un membre se fait sous la forme suivante :

structure **.** *membre*

- *structure* désigne la variable structurée
- *membre* désigne la variable membre

Exemples.

– *déclar. de types :*

```
struct date {short int jour, mois, annee;};
struct entete_commande
      { char code_type_ligne;   /* 'C' */
        unsigned int no_commande, no_client;
        struct date date_commande; };
struct ligne_commande
      { char code_type_ligne;   /* 'L' */
        unsigned int no_commande,
                     no_produit, qte_commande; };
struct bon_commande
      { struct entete_commande entete;
        struct ligne_commande corps[15]; };
```

– *déclar. de variables :*

```
struct bon_commande commande_en_cours,
                    commande_archivee;
```

– *désign. de membres :*

```
commande_en_cours.entete
commande_en_cours.entete.no_commande
commande_archivee.entete.date_commande
commande_archivee.entete.date_commande.annee
commande_en_cours.corps[i]
commande_en_cours.corps[i].no_produit
commande_en_cours.corps[i].qte_commande
```

Ces exemples le montrent : l'opérateur **.** de sélection d'un membre est associatif.

Exemple complet

```
int date_valide (struct {short int jour,mois, annee;} date)
      /* test de la validité syntaxique d'une date */
      /* N.B. Les mois sont donnés dans l'intervalle [1..12] */
{
  extern short jours_mois (short mois, short annee);
                                /* --> nbre. jours du mois */
  return ( (date.mois >= 1 && date.mois <= 12)
        && (date.jour >= 1
         && date.jour <= jours_mois(date.mois,date.annee)) );
}
```

- ***Portée des déclarations***

Un nom de membre n'est jamais employé seul; il est toujours préfixé par le nom de la variable structurée dont il fait partie et c'est cette désignation complète qui doit être univoque. En conséquence :

- tous les noms de membres d'une même structure doivent être différents;
- le même nom peut identifier des membres de deux structures distinctes (ex.: le membre *no_commande* des structures *entete_commande* et *ligne_commande*).

Opérations (3). Copie de structures

Il est possible de **copier** des structures, c'est-à-dire d'affecter à une variable structurée S_2 une autre variable structurée S_1 *du même type*. Cette copie peut prendre trois formes :

- expression d'**affectation** absolue (=) : ex.:
```
commande_archivee = commande_en_cours;
date_paiement = date_du_jour();
```
- affectation à un **paramètre** de fonction,
- affectation au **résultat** d'une fonction.

Exemple. La fonction *echeance()* reçoit en paramètre une date de départ et un délai *n* en nombre de jours; elle rend en résultat la date d'échéance à "n jours fin de mois" (n = 30, 60, 90 ...). Les deux dates, paramètre et résultat, sont deux variables structurées du même type. Puisque le paramètre formel *date de départ* est une copie du paramètre effectif, la préparation du résultat peut se faire par modification du paramètre formel, plutôt que d'utiliser une variable de manoeuvre.

echeance.c

```
struct date {short jour, mois, annee;};
short jours_mois(short mois,short annee); /*-> nbre.jours du mois*/

struct date echeance (struct date date_d, short delai)
{
 date_d.mois += delai / 30;                    /* mois d'échéance */
             /* si mois > 12, corriger l'année et le mois : */
 date_d.annee += (date_d.mois - 1) / 12;
 date_d.mois = 1 + ((date_d.mois - 1) % 12);
 date_d.jour = jours_mois(date_d.mois,date_d.annee); /* fin mois */
 return (date_d);
}
```

Initialisation des variables structurées

La déclaration d'une variable structurée peut comporter une clause d'initialisation.

• ***Initialisation constante***

Cette initialisation peut toujours se faire par une liste d'expressions constantes, dont chacune indique la valeur initiale d'un membre, en suivant l'ordre de déclaration des membres.

Exemple : `struct date  date_de_reference = {27, 11, 1993};`

Si un membre est lui-même une variable composite (structure ou tableau), ses valeurs composent une sous-liste entre accolades.[47]

```
Exemple :     /* types : */
      struct raison_sociale {char categorie[9], nom[25];};
      struct localite {short int no_postal; char nom[17];};
      struct adr_postale {char rue_no[31]; struct localite loc;};
            /* struct constante = variable initialisée : */
      const struct {struct raison_sociale ident;
                    struct adr_postale adr;}
           etiquette = { { "I.E.S.", "Saint-Laurent"},
                         { "Quai Mativa  38", { 4020, "Liège" } }
                       } ;
```

La liste d'initialisation peut comporter moins de valeurs qu'il n'y a de membres déclarés; dans ce cas, seuls les premiers membres sont initialisés. Comme dans tous les cas d'initialisation incomplète, les autres membres sont initialisés à la valeur 0.

Exemple : `struct bon_commande  commande_en_cours = {'C'};`
cette déclaration initialise uniquement le code_type_ligne de l'entete_commande

• ***Initialisation variable***

L'initialisation d'une variable structurée *automatique* peut se faire par une **copie** de structure.

```
Exemples :    struct date  date_du_jour(void); /* fonction -> date */
              struct date  date_commande = date_du_jour(),
                           date_facture = date_commande;
```

1.4. Types alternatifs : les unions

Une variable de type union est une zone de mémoire dont le contenu peut être interprété de différentes manières. Chacune des interprétations possibles est déclarée comme un membre de l'union.

Déclaration

Mis à part le nom de la méthode de construction, la déclaration d'un type d'union est identique à celle d'un type de structure.

[47] Pourvu que la sous-liste soit complète, elle peut ne pas être placée entre accolades (cf. chapitre 7).

union *étiquette_de_type* { *liste des membres* } ;

Exemple : Un programme *calculette* doit analyser et exécuter une expression arithmétique introduite au clavier. L'analyse de cette expression en isole les différents symboles – *nombres* (convertis en `float`), *opérateurs* ou *noms* de fonctions – et les range successivement dans une même zone de mémoire.

– *déclar. du type :* `union symbole {float nbre; char nom[32]; char oper;};`
ou : `union symbole {float nbre; char nom[32], oper;};`
– *déclar. de variable :* `union symbole symbole_lu;`

La clause d'**initialisation** éventuellement incorporée à la déclaration d'une variable union ne peut évidemment inscrire qu'une seule valeur dans cette variable; le type de cette valeur initiale doit être celui du *premier membre* de l'union.

Exemple :
```
union symbole  symbole_lu = 0;
    /* valeur initiale du membre 'nbre' - type float */
```

Utilisation

Une variable union doit toujours être utilisée en association avec une autre variable qui indique quelle interprétation doit en être faite. Le plus souvent, ces deux variables sont associées comme membres d'une même structure.

Exemple : Le programme *calculette* crée un descripteur de chaque symbole. Cette structure descriptive comporte un code de *catégorie*, l'indication de la *longueur* et le *texte* du symbole; celui-ci est soit un *nombre* (converti en `float`), soit un *nom* de fonction, soit un *opérateur* d'un seul caractère. L'interprétation du *texte* est dictée par le code de *catégorie*.

```
enum categ_symbole {NBRE='9', NOM ='Z', OPER='+'};
struct descr_symbole { enum categ_symbole categ;
                       int longueur;
                       union { float nbre;
                               char nom[32];
                               char oper;    } texte; };
```

Pour produire le même effet, on peut déclarer une union de structures qui contiennent toutes à la même position le préfixe identificateur.

Exemple :
```
struct descr_nbre { enum categ_symbole categ;
                    int longueur;
                    float texte;              };
struct descr_nom  { enum categ_symbole categ;
                    int longueur;
                    char texte[32];           };
struct descr_oper { enum categ_symbole categ;
                    int longueur;
                    char texte;               };
union descr_symbole { struct descr_nbre nbre;
                      struct descr_nom nom;
                      struct descr_oper oper; };
```

Opérations

Une union peut être assimilée à une structure dont tous les membres occuperaient la même position.

Les opérations permises sur une variable union sont les mêmes que les opérations possibles sur une structure.

Exemples

Soit les déclarations :
```
struct descr_symbole {enum categ_symbole categ;
                      int longueur;
                      union {float nbre;
                             char nom[32];
                             char oper;    } texte;};
struct descr_symbole  symb_courant, symb_suivant;
```

• **désignation d'un membre** (**.**) : `symb_courant.texte.oper`
`symb_suivant.texte.nom`

• **taille** (**`sizeof`**) : `sizeof(symb_courant.texte)`
donne la taille du membre le plus long

• **affectation** (**=**) d'une variable du même type : `symb_courant.texte = symb_suivant.texte;`
• paramètre ou résultat d'une **fonction** :

1.5. Les champs de bits

On a déjà noté au chapitre 4 que certaines informations peuvent être représentées dans un espace de mémoire plus petit qu'un octet : ***un espace de* n *bits suffit pour représenter* 2^n *informations mutuellement exclusives***.

Exemple

Pendant qu'il analyse le texte d'un programme, un compilateur constitue une table des déclarations qu'il rencontre. Un compilateur C pourrait coder de la manière suivante les attributs d'une déclaration de variable (ou résultat de fonction) scalaire :

bits :

0 :	mode de déclar. :	0 = interne, 1 = extern
1- 2 :	portée et nature :	00 = fonction globale, 01 = variable globale 10 = paramètre (local à une fonction), 10 = variable locale (à un bloc)
3- 4 :	droit de mise à jour :	00 = par défaut, 01 = const, 10 = volatile, 11 = const volatile
5- 6 :	classe :	00 = static, 10 = auto, 11 = register
7- 8 :	mode de représentation :	00 = entier non signé, 01 = entier signé, 11 = réel (signé)
9-10 :	taille = 2^n octets :	n = 0, 1, 2, 3
11 :	initialisation :	0 = clause absente, 1 = clause présente

total : 12 bits

Chaque groupe de bits ainsi défini constitue un **champ de bits** adjacents à l'intérieur d'une variable. Il existe deux manières de traiter ces informations.

Une première manière est de définir une variable entière (pour l'exemple ci-dessus, cette variable devrait être de taille `short`, elle contiendrait 16 bits) et d'agir sur les groupes de bits adjacents au moyen d'opérations logiques de masquage.[48]

L'autre manière est de définir une variable structurée dont les membres sont les différents champs de bits.

Déclaration

`struct` *étiquette_de_type* `{` *liste des champs* `} ;`

Un champ de bits est déclaré comme ceci :

[*type* *identificateur*] **:** *longueur*

Le *type* doit être [`un`]`signed int` .
La *longueur* est une expression entière constante indiquant le nombre de bits.

Si l'on mentionne seulement la longueur (après les **:**), le champ existe mais n'est pas référençable.

Exemple. L'exemple ci-dessus pourrait donner lieu aux déclarations suivantes :

– *déclar. de type :*

```
struct attributs { unsigned mode:1, portee:2,
                                droit:2, classe:2,
                                type:2, taille:2,
                                val_initiale:1;   } ;
```

– *déclar. de variable :*

```
        /* tableau de 120 déclarations : */
struct { char nom [31+1];
         struct attributs attrib; }
       declar [120];
```

– *désign. d'un champ :*

```
        /* enregistrer une déclar. de 'float' : */
declar[i].attrib.type = 3;    /* réel signé */
declar[i].attrib.taille = 2;  /*  2 exp. 2  */
```

Utilisation

Chaque champ de bits se comporte comme un `int`, signé ou non, selon la déclaration, et dont la longueur (en nombre de bits) est précisée.

REMARQUE. Chaque compilateur C précise ses propres conditions d'utilisation des champs de bits, notamment la longueur totale de la structure (ainsi, pour Turbo C, toute structure de bits est un `int`) et l'ordre de succession des bits (de droite à gauche ou de gauche à droite).

[48] Cf. chapitre 4.

2. Déclaration de noms de types

Pour tout type de données – type prédéfini ou type construit –, le programmeur peut déclarer un ou plusieurs noms ou identificateurs synonymes. Partout où doit être référencé un type de données (dans une déclaration de variables ou de fonctions, en guise d'opérateur de coercition ou comme opérande de *sizeof*), il peut être désigné par un de ses synonymes.

Déclaration et utilisation

La déclaration de synonymes prend la forme suivante :

```
typedef   spécification_de_type   identificateur ,
                                  identificateur ,
                                  ..... ;
```

Ce format est le même que celui des déclarations de variables, le mot `typedef` prenant la place de l'indication de classe.

Exemples

– *déclar. de synonymes :*
```
typedef signed int  Quantite;
typedef float  Prix;
typedef unsigned int  Numero;
typedef unsigned int  size_t;
```
[49]

– *déclar. de variables :*
```
Prix  prix_net, a_payer;
Quantite  qte_en_stock, stock_plancher;
Quantite  qte_commandee, qte_livree;
Numero  no_client, no_produit;
```

– *déclar. de type :*
```
enum valeur_logique {FAUX, VRAI};
```
– *déclar. de synonyme :*
```
typedef enum valeur_logique  Valeur_Logique;
```
– *déclar. combinée :*
```
typedef enum {FAUX, VRAI}  Valeur_Logique;
```
– *déclar. de variables :*
```
Valeur_Logique  option = VRAI, reponse;
```

– *déclar. de type :*
```
struct date {short int jour, mois, annee;};
```
– *déclar. de synonyme :*
```
typedef struct date  Date;
```
– *déclar. combinée :*
```
typedef struct {short int jour, mois, annee;}
        Date;
```
ou :
```
typedef struct date {short int jour,mois,annee;}
        Date;
```
– *déclar. de variables :*
```
Date  date_echeance, date_paiement;
```
– *déclar. de fonction :*
```
Date  date_du_jour(void);
```

[49] La déclaration de `size_t` (*"size type"*) fait partie de la bibliothèque standard du langage C. Elle définit le format des indications de taille manipulées par les fonctions standards.

– *déclar. de types :*
```
/* en-tête de commande : */
typedef struct { char code_type_ligne;
                 Numero no_commande, no_client;
                 Date date_commande; }
       Entete_Commande;
  /* ligne de corps : */
typedef struct { char code_type_ligne;
                 Numero no_commande, no_produit;
                 Quantite qte_commande; }
       Ligne_Commande;
  /* bon de commande: en-tête + 15 lignes max. */
typedef struct { Entete_Commande entete;
                 Ligne_Commande corps[15]; }
       Bon_Commande;
```
– *déclar. de variables :* `Bon_Commande commande_en_cours,commande_archivee;`

Il est permis de déclarer un synonyme pour un *type défini ultérieurement*. Exemple :

– *déclar. de synonyme :* `typedef struct date  Date;`
– *déclar. de type :* `struct date {short int jour, mois, annee;};`

Portée des déclarations

Comme pour la déclaration d'une variable, l'endroit du texte où est rédigée la déclaration d'un type définit la portée, globale ou locale, de cette déclaration.

La déclaration d'un type doit être visible à tout endroit du texte où ce type est référencé, en particulier à tout endroit où on déclare une variable ou une fonction possédant ce type.

REMARQUE

Il existe une différence importante entre les ***étiquettes*** de types construits et les ***identificateurs*** de types déclarés par `typedef` : alors qu'une étiquette de type n'est jamais employée seule, mais est toujours précédée du nom de la méthode de construction du type, un identificateur de type est employé seul.

Exemple : dans la déclaration de type combinée ci-dessous :
```
typedef struct date {short int jour, mois, annee;}  Date;
```
date est une étiquette et *Date* est un identificateur;
les deux déclarations de variables qui suivent sont synonymes :
```
struct date  date_echeance;
Date  date_echeance;
```

Cet exemple montre l'utilité principale des identificateurs synonymes de types : la simplification d'écriture et la lisibilité.

Une certaine tradition veut que, pour les distinguer des identificateurs de variables ou fonctions (habituellement écrits en minuscules) et des identificateurs de constantes (souvent écrits en majuscules), les identificateurs de types soient écrits en minuscules avec initiale majuscule (voir les exemples ci-dessus).[50]

[50] Noter qu'il est licite d'écrire ce qui suit :
```
typedef struct date {short int jour, mois, annee;} date;
struct date date_echeance;    ou    date date_echeance;
```

Exercices

struct – enum

1.

Impression d'étiquettes d'adresses

Définir le tableau des fiches signalétiques des étudiants d'une classe.
Chaque fiche contient les informations suivantes :

- numéro d'inscription (type entier),
- nom et prénom,
- sexe (codé par l'initiale Masculin, Féminin),
- état civil (codé par l'initiale Célibataire, Marié, Divorcé, Veuf),
- adresse postale :
  - rue et numéro,
  - numéro de boîte,
  - localité :
    - numéro du code postal,
    - nom

Remarque. Les deux codifications (sexe et état civil) sont à déclarer par `enum`.

Programmer les fonctions suivantes.

• ***Garnir le tableau ← Garnir une fiche***
(Définir judicieusement les paramètres et le résultat de la fonction *Garnir une fiche*.)
Le texte de chaque fiche est présenté sur une ligne dans un fichier d'entrée à lire par *scanf()*; chaque donnée y est suivie du signe ";".

• ***Imprimer les étiquettes d'adresses ← Imprimer une étiquette***
(Définir judicieusement les paramètres et le résultat de la fonction *Imprimer une étiquette*.)
Les étiquettes sont écrites dans un fichier de sortie.
Chaque étiquette est mise en page dans une zone rectangulaire de 8 lignes de 40 colonnes.
Chaque étiquette présente les informations suivantes :

- coin supérieur gauche : numéro d'inscription;
- ligne d'identification : préfixe "Mademoiselle/Madame/Monsieur" + nom et prénom;
- 1ère ligne d'adresse : rue et numéro, préfixe "Bte" + numéro de boîte;
- 2ème ligne d'adresse : numéro du code postal, nom de la localité.

Créer le programme et le tester.

typedef

2.

Récrire le programme ci-dessus en déclarant les types par `typedef`.

struct – enum – union

3.

Déclarer le type ***prix*** répondant à la définition suivante : un *prix* est composé d'un code de *devise* ('$' ou 'F') et d'un *montant*. Un montant exprimé en dollars (code '$') est un nombre réel, un montant exprimé en francs belges (code 'F') est un nombre entier long.

Chapitre 9. Les pointeurs

1. Concepts de base

Supposons que nous voulions écrire une fonction permuter() *chargée de permuter en mémoire deux variables de même type, par exemple : deux éléments d'un tableau. On ne peut appeler cette fonction en lui passant en paramètres les variables à permuter; en effet, une fonction appelée reçoit une* copie *de ses paramètres et ne dispose d'aucun moyen pour modifier la valeur des paramètres effectifs dans la fonction appelante. La solution consiste à utiliser comme paramètres des* ***pointeurs****.*

1.1. Définitions

- Un **pointeur** est une variable qui, en guise de valeur, contient l'adresse d'une autre variable.

  Soit un pointeur `p` contenant l'adresse d'une variable `v`;
  on dit que "`p` *pointe sur* `v`" et que "`v` est *repéré* par `p`".

- Aux pointeurs sont nécessairement associés deux **opérateurs d'adressage** :

  `&` l'expression **`&v`** fournit l'adresse de `v`, qu'on peut affecter à `p` en programmant : `p = &v;`

  Puisque l'adresse de `v` est immuable et connue du compilateur,
  l'expression `&v` est une ***expression constante***, de type "adresse".[51]
  Comme `sizeof`, l'opération **`&`** *est une opération évaluée par le compilateur.*

  `*` l'expression **`*p`** désigne *indirectement*, non pas le pointeur `p`, mais l'objet `v` sur lequel il pointe (* sert ici d'opérateur d'**indirection**).

- La **déclaration** d'un pointeur a la forme d'une déclaration de variable ordinaire,[52] à ceci près que :

  – le type indiqué est celui de l'objet repéré par le pointeur (qui est lui-même du type "adresse");

  – l'identificateur attribué au pointeur est préfixé par * (imitation de l'opération d'indirection);

  – cet identificateur peut être précédé des qualificatifs *const* et *volatile*
  modifiant le droit d'accès au pointeur (pas le droit d'accès à l'objet repéré).

exemples :

```
short int v;      /* variable de type short int */
short int * p=&v;
      /* pointeur sur un objet de type short int
         initialisé pour pointer sur l'objet v */
```

[51] L'opération `&` (adresse de) n'a aucun sens (et elle est interdite) avec une variable de classe `register` ou un champ de bits dans une structure de bits. Si `v` est le nom d'un tableau ou le nom d'une fonction, l'opérateur `&` est facultatif.
[52] Cf. chapitre 7.

on peut écrire :
```
short int v, t[12], * const p=&v, * q;
/* v        scalaire
   t        tableau
   p        pointeur constant initialisé
   q        pointeur variable non initialisé */
```

Illustration des définitions

	id	*adr*	*val*
`short int v = 0;`	v	16	0
`short int * p;`	p	18	
	id	*adr*	*val*
	v	16	0
`p = &v;`	p	18	**16**
	id	*adr*	*val*
`*p += 1;`	v	**16**	1
	p	18	16

1.2. Les pointeurs comme paramètres de fonctions

Revenons à la fonction permuter(). *Elle sera appelée avec deux paramètres effectifs : les* adresses *des deux variables à permuter. Comme cela se produit pour tous les paramètres, ces adresses sont copiées dans les paramètres formels correspondants de la fonction appelée; ces paramètres formels sont donc des pointeurs. En conséquence, les opérations de la fonction appelée peuvent, par le biais d'un adressage indirect, modifier le contenu des variables locales de la fonction appelante.*

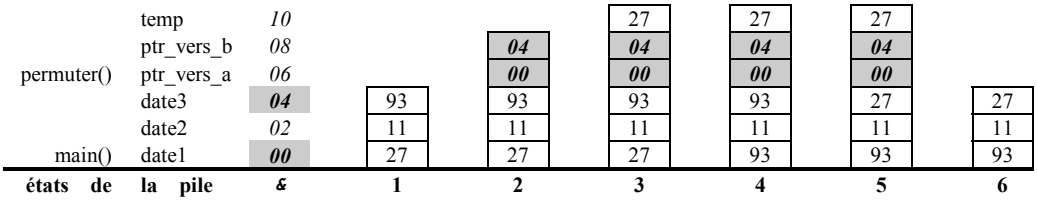
test.c

```
#include <stdio.h>

void permuter (int* ptr_vers_a, int *ptr_vers_b);
void main(void)          /* test de la fonction permuter() */
{
   int date1, date2, date3;
   printf ("\nDonnez une date jj/mm/aa : ");
   scanf ("%2d/%2d/%2d", &date1, &date2, &date3);                    1
   permuter(&date1, &date3);      /* & --> adresse de */             6
   printf (" --> aa/mm/jj = %2d/%2d/%2d", date1, date2, date3);
}
```

permuter.c

```
void permuter (int * ptr_vers_a, int*ptr_vers_b)                    2
   /* permute en mémoire les entiers a et b */
{
   int temp = *ptr_vers_a;           /* sauver la valeur de a */    3
   *ptr_vers_a = *ptr_vers_b;  /* copier dans a la valeur de b */   4
   *ptr_vers_b = temp; /* copier dans b l'ancienne valeur de a */   5
}
```



2. Pointeurs et variables composites

2.1. Variables composites et adressage indirect

• On peut prendre l'**adresse d'une variable composite** – structure ou union – ou l'**adresse d'un membre**, et la placer dans un pointeur. L'adresse d'un membre est obtenue par une expression de la forme **`&var.m`** .

L'opérateur **`.`** étant davantage prioritaire que **`&`**, `&var.m` ⇔ `&(var.m)` .

Exemples :

```
      /* déclaration de types composites : */
struct date {char jj, mm, aa;};
            /* char : entier de petite taille */
struct tete_commande { unsigned int no_commande;
                       struct date date_reception;
                       unsigned int no_client; };
struct ligne_commande { unsigned int no_commande;
                        unsigned int no_article;
                        signed int qte_commande; };
union bon_commande { struct tete_commande tete;
                     struct ligne_commande ligne; };
      /* déclaration d'une variable composite : */
union bon_commande  en_cours;
      /* déclaration/initialisation de pointeurs : */
union bon_commande  * pt_bon = &en_cours;
struct tete_commande  * pt_tete = &en_cours.tete;
struct ligne_commande  * pt_ligne = &en_cours.ligne;
struct date  *pt_date = &en_cours.tete.date_reception;
unsigned int  *pt_article = &en_cours.ligne.no_article;
```

déclarations équivalentes :

```
      /* déclaration de types composites : */
typedef struct {char jj, mm, aa;}  Date;
            /* char : entier de petite taille */
typedef struct { unsigned int no_commande;
                 Date date_reception;
                 unsigned int no_client; }
        Tete_Commande;
typedef struct { unsigned int no_commande;
                 unsigned int no_article;
                 signed int qte_commande; }
        Ligne_Commande;
typedef union { Tete_Commande tete;
                Ligne_Commande ligne; }
        Bon_Commande;
      /* déclaration d'une variable composite : */
Bon_Commande  en_cours;
      /* déclaration/initialisation de pointeurs : */
Bon_Commande  * pt_bon = &en_cours;
Tete_Commande  * pt_tete = &en_cours.tete;
Ligne_Commande  * pt_ligne = &en_cours.ligne;
Date  *pt_date = &en_cours.tete.date_reception;
unsigned int  *pt_article = &en_cours.ligne.no_article;
```

• Il est possible de déclarer un pointeur pour un **type construit défini ultérieurement**.

Exemple :
```
struct date * pt_date;
struct date {char jj, mm, aa;};
```

• Comment **accéder indirectement à un membre** d'une variable composite repérée par un pointeur ?

On peut utiliser une expression de la forme `(*pointeur).membre`
– l'opérateur `.` étant davantage prioritaire que `*`, il est nécessaire d'écrire les parenthèses.

Il existe une écriture simplifiée : **`pointeur->membre`** ⇔ `(*pointeur).membre`
– la flèche est formée des deux caractères "-" et ">" sans espace entre eux.

Exemples équivalents :
```
pt_date->aa
      ⇔ (*pt_date).aa
pt_tete->date_reception.aa
      ⇔ (*pt_tete).date_reception.aa
pt_bon->tete.date_reception.aa
      ⇔ (*pt_bon).tete.date_reception.aa
```

• **N.B.** On peut **comparer** entre eux deux pointeurs ou adresses repérant des objets quelconques à l'intérieur d'une même variable composite.

Exemples : ces relations sont toutes vraies :
```
pt_bon == pt_ligne
pt_ligne >= pt_tete
pt_tete < pt_date
&en_cours.tete != &(pt_tete->no_client)
```

2.2. Les variables composites comme paramètres de fonctions

Au lieu de passer en paramètre à une fonction une (copie d'une) structure, il est souvent plus intéressant de lui passer l'*adresse* de cette structure. On évite ainsi de recopier la structure (qui peut être longue) dans un paramètre formel; si la fonction modifie la structure, on évite de devoir encore la recopier en retour.

Exemple: fonction permutant les éléments d'une date : *jj,mm,aa* ↔ *aa,mm,jj* .

```
typedef struct {short d[3];} Date_Num;
Date_Num date_permutee (Date_Num date)
{
  /* modifier le paramètre formel (copie) en adressage direct : */
   date.d[0] ^= date.d[2] ^= date.d[0] ^= date.d[2];
   return (date);        /* transférer la copie modifiée */
}
```

```
typedef struct {short d[3];} Date_Num;
void date_permutee (Date_Num * pt_date)
{
 /* modifier le paramètre effectif en adressage indirect : */
  pt_date->d[0] ^= pt_date->d[2] ^= pt_date->d[0] ^= pt_date->d[2];
}
```

3. Pointeurs et tableaux

3.1. Parenté entre tableaux et pointeurs

Soit une déclaration de tableau,[53] par exemple : `short int t[N];`

> Lorsqu'on le suffixe par un indice, le nom `t` désigne un élément du tableau : `t[i]`.
> ***Employé seul, le nom* `t` *désigne l'*adresse *de début du tableau :* `t ⇔ &t[0]`**
> *(* `t` *n'est donc pas synonyme de* `t[0]` *).*
>
> **En d'autres termes, un identificateur de tableau désigne une *constante*** (contenant une) ***adresse*,**
> *sauf dans les opérations du compilateur* `sizeof` *et* `&`*, où il désigne le tableau;*
> il s'ensuit que : **`t ⇔ &t`**
> `sizeof(t)` donne la taille du tableau, pas de la constante adresse.

val	**76**	variable	variable	variable	variable	variable	variable
adr		*76*	*78*	*80*	*82*	*84*	*86*
désign	**`t`**	`t[0]`	`t[1]`	`t[2]`	`t[3]`	`t[4]`	`t[5]`
désign		`*t`	`*(t+1)`	`*(t+2)`	`*(t+3)`	`*(t+4)`	`*(t+5)`

Puisque le nom `t` désigne en réalité (un mot de mémoire contenant) une adresse, les deux modes de désignation illustrés ci-dessus, par indexation `t[i]` ou par indirection `*(t+i)`, sont équivalents.[54]

> **Note.** C'est cette équivalence qui justifie que les valeurs d'indice commencent à 0 plutôt qu'à 1.

3.2. Calculs d'adresses par déplacement

Cette équivalence explique les règles du calcul d'adresse : augmenter (ou diminuer) un pointeur d'un "**déplacement**" `i` ajoute (ou soustrait) au nombre-adresse qu'il contient `i` *fois la taille (sizeof) du type d'objet repéré*. Le ***déplacement unitaire*** est égal à la taille du type d'objet repéré.

> *Exemple :* soit la déclaration `short int *p;`
> si `p` contient l'adresse 76, `++p` a pour résultat 78 (si `sizeof(short)==2`)

Pour d'évidentes raisons de cohérence, la même règle s'applique aux *constantes adresses*.

> *Exemple :* soit la déclaration `short int t[12];`
> si le tableau est implanté à partir de l'adresse 76, `&t[0]` vaut 76, `&t[0]+1` vaut 78

---

[53] Cf. chapitre 2.

[54] L'équivalence est poussée jusqu'à autoriser l'indexation d'un nom de pointeur et l'indirection via un nom de tableau !
Soit la déclaration : `int i, t[N], *p=&t;  /* i=entier, t=tableau, p=pointeur */`
Les désignations suivantes sont autorisées et équivalentes : `t[i]  *(t+i)  *(p+i)  p[i]`
Puisque `t[i] ⇔ *(t+i) = *(i+t)`, il est même permis d'écrire `i[t]` !
Puisque `t ⇔ &t`, on peut aussi écrire `(&t)[i] ⇔ t[i]`, différent de `&t[i] ⇔ &(t[i])` ...

Le programme ci-dessous illustre cette règle de calcul.

```
#include <stdio.h>

void main(void)
{
  float fl[4], * p_sur_fl = fl;     /* pointe sur fl[0] */
      /* taille de l'élément sur lequel on pointe : */
  printf("\n\n sizeof(float) = %d",sizeof(*p_sur_fl));
      /* contenu du pointeur converti en entier non signé,
         affiché en hexadécimal : */
  printf("\n p_sur_fl   vaut %x",(unsigned int)p_sur_fl);
  printf("\n p_sur_fl+1 vaut %x",(unsigned int)++p_sur_fl);
  printf("\n &fl[0]     vaut %x",(unsigned int)&fl[0]);
  printf("\n &fl[0]+1   vaut %x",(unsigned int)(&fl[0]+1));
}
```

Programme équivalent, utilisant le format "`%p`" d'affichage d'un pointeur.

```
#include <stdio.h>

void main(void)
{
  float fl[4], * p_sur_fl = fl;     /* pointe sur fl[0] */
      /* taille de l'élément sur lequel on pointe : */
  printf("\n\n sizeof(float) = %d",sizeof(*p_sur_fl));
      /* "%p" affiche le contenu d'un pointeur : */
  printf("\n p_sur_fl   vaut %p",p_sur_fl);
  printf("\n p_sur_fl+1 vaut %p",++p_sur_fl);
  printf("\n &fl[0]     vaut %p",&fl[0]);
  printf("\n &fl[0]+1   vaut %p",&fl[0]+1);
}
```

Les *calculs d'adresses* n'ont de sens qu'entre des pointeurs ou adresses repérant des éléments d'un même tableau. Par définition, tous les éléments d'un tableau ont le même type; le déplacement unitaire est donc défini.

Il est permis d'englober dans ces calculs la *position suivant immédiatement le dernier élément du tableau*. Cette tolérance a pour but de rendre possibles les tests de détection de la fin du tableau.

Dans les limites ainsi fixées, les opérations suivantes sont possibles.

Déplacement

On peut modifier la valeur d'une constante adresse ou d'un pointeur repérant un élément d'un tableau, en ajoutant ou soustrayant une expression entière.

Exemples : `p-1` `t+(dimension/2)`

La valeur de l'expression entière est automatiquement multipliée par la taille de l'élément repéré, de manière à donner un déplacement d'adresse.

Soustraction de deux pointeurs ou adresses

On peut effectuer une soustraction entre deux pointeurs ou constantes adresses repérant deux éléments d'un même tableau.

Exemple : `dimension = 1 + ptr_fin - ptr_debut`

Le résultat est une valeur entière signée égale à la différence des adresses de départ divisée par la taille du type d'élément repéré. Autrement dit, le résultat est égal à la différence des indices correspondants.

Remarque. Soit trois pointeurs sur trois éléments d'un tableau : `debut`, `fin`, `milieu`. On peut calculer `milieu=debut+(fin-debut)/2`; il est interdit d'écrire : `milieu=(debut+fin)/2`, car l'addition de deux pointeurs `(debut+fin)` ne donnerait pas une adresse pointant sur un objet connu du programme.

Comparaison de deux pointeurs ou adresses

Toutes les comparaisons sont possibles entre deux pointeurs ou adresses repérant deux éléments d'un même tableau. Le résultat est le même que si l'on comparait les indices correspondants.

Exemple : `ptr_debut <= ptr_fin`

Illustration

```
short int t[6], *p, *q;     /* sizeof(short int) == 2 */
.....
p = t;       ⇔     p = &t[0];
q = p+5;     ⇔     q = &t[5];
                                              q - p  →  5
                                              p - q  → -5
                                    p <= q
                                    q == p+5
```

val	**76**							
adr		*76*	*78*	*80*	*82*	*84*	*86*	
désign	**t**	`t[0]`	`t[1]`	`t[2]`	`t[3]`	`t[4]`	`t[5]`	
désign		`*p`	`*(p+1)`	`*(p+2)`	`*(p+3)`	`*(p+4)`	`*(p+5)`	
désign		`*(q-5)`	`*(q-4)`	`*(q-3)`	`*(q-2)`	`*(q-1)`	`*q`	

ATTENTION. L'opérateur ***** d'indirection est davantage prioritaire que les opérateurs d'addition et soustraction. En conséquence :

`*p+1` ⇔ `(*p)+1`	augmente de 1 la valeur de l'objet dont l'adresse est **p*
`*(p+1)`	désigne, dans un tableau, l'élément qui suit l'élément d'adresse **p*

3.3. Les tableaux comme paramètres de fonctions

Si l'on donne comme paramètre effectif à un appel de fonction l'identificateur d'un tableau, sans indice, on passe comme paramètre l'*adresse* de ce tableau. Cette adresse est copiée dans le paramètre formel correspondant de la fonction appelée.

Ce paramètre formel est donc un pointeur. Il peut être déclaré comme un pointeur `*p` ordinaire ou comme un tableau `t[]` de dimension indéterminée.

Exemple: la fonction standard *strcpy()* copie une chaîne de caractères *src* dans un tableau *dest*
et renvoie l'adresse du tableau *dest*;
elle pourrait notamment être programmée d'une des deux manières suivantes :

```
char* strcpy (char* ptr_dest, const char *ptr_src)
{
   char * ptr_dst = ptr_dest; /* mémoriser l'adr. de destination */
   while (*ptr_src)           /* pour tous les caractères != \0 */
   { *ptr_dest++ = *ptr_src++; }  /* copier le caractère courant
                                     puis incrémenter les pointeurs */
   *ptr_dest = *ptr_src;      /* copier le terminateur */
   return (ptr_dst);    /* rendre l'adresse de destination */
}
```

```
char *strcpy (char dest[], const char src[])
{
   int i;                           /* indice */
   for (i=0;src[i];++i)   /* pour tous les caractères != \0 */
   { dest[i] = src[i]; }     /* copier le caractère courant */
   dest[i] = src[i];      /* copier le terminateur */
   return (dest);       /* rendre l'adresse de destination */
}
```

Lorsqu'on crée une fonction manipulant un tableau, normalement, on désire qu'elle soit capable de traiter un tableau de n'importe quelle dimension. Pour cette raison, comme dans le dernier exemple ci-dessus, on déclare alors le paramètre formel comme un tableau de dimension indéterminée : `t[]` .

De plus, il est nécessaire de fournir à la fonction une information lui permettant de connaître, à chaque appel, la dimension effective du tableau dont elle reçoit l'adresse. Ou bien, comme dans l'exemple ci-dessus, le tableau contient un élément terminateur (c'est toujours le cas d'une chaîne de caractères : *\0*), ou bien un paramètre supplémentaire indique explicitement la dimension du tableau.

Exemple: la fonction standard *memcpy()*
copie dans un tableau *dest* les *n* premiers octets d'un tableau *src*
et renvoie l'adresse du tableau *dest*;
elle pourrait notamment être programmée d'une des deux manières suivantes :

```
char * memcpy (char dest[], const char src[], unsigned int dimens)
{
   unsigned int i;                                  /* indice */
   for (i=0; i < dimens; ++i) dest[i] = src[i];
   return (dest);       /* rendre l'adresse de destination */
}
```

```
char* memcpy(char*pt_dest, const char *pt_src, unsigned int dimens)
{
   char*pt_dst = pt_dest;    /* mémoriser l'adr. de destination */
   while (dimens--) *pt_dest++ = *pt_src++;
   return (pt_dst);          /* rendre l'adresse de destination */
}
```

3.4. Parenté entre chaînes de caractères et pointeurs

On l'a déjà dit, une chaîne de caractères littérale (par exemple, `"Bonjour \n"`) est un *tableau de caractères constant*. Elle possède toutes les propriétés d'un tableau ordinaire, à deux exceptions près : (1) il est impossible d'en modifier le contenu; (2) elle ne possède pas de nom identificateur.

Lorsque le compilateur rencontre dans le texte du programme une chaîne de caractères littérale, il stocke le texte (prolongé d'un terminateur *\0*) dans un tableau sans nom et manipule en réalité l'*adresse* de ce tableau.

Les seuls emplois possibles d'une chaîne de caractères littérale sont :

- l'affectation de son adresse à un paramètre de fonction (emploi fréquent) :

  ex.: l'appel `printf("Bonjour \n")` passe en paramètre l'*adresse* du texte "Bonjour \n" le paramètre formel correspondant dans la fonction *printf()* est un pointeur

- l'affectation de son adresse à un pointeur nommable (peu utile) :

```
ex.:    char* p_msg;
        .....
        p_msg = "Bonjour \n";
```

- l'initialisation d'un pointeur lors de sa déclaration (manière détournée de rendre la constante nommable) :

```
ex.:    char * p_msg = "Bonjour \n";
```

- l'initialisation d'un tableau de caractères :

```
ex.:    char t_msg[] = "Bonjour \n";
```

  Ce dernier emploi est le seul qui *copie le texte plutôt que son adresse*. Le tableau `t_msg[]` est ajusté à une taille suffisante pour contenir le texte et son terminateur `\0`; le texte est copié dans ce tableau, et la chaîne de caractères littérale n'est pas créée par ailleurs. Autre différence par rapport à l'exemple précédent : le contenu du tableau est modifiable.

Sauf dans ce dernier cas, **lorsque l'on cite une chaîne de caractères littérale, on manipule son *adresse*.**

Les instructions ci-dessous produisent donc le même effet :

```
printf(&"Bonjour \n");        printf(p_msg);
printf("Bonjour \n");         printf(t_msg);
```

3.5. Note sur les pointeurs comme résultats de fonctions

La discussion sur l'utilisation des chaînes de caractères va nous permettre d'illustrer une règle importante.

> **Si une fonction renvoie pour résultat un pointeur ou une adresse, l'objet repéré doit être :**
>
> - soit une **constante littérale**;
> - soit un **objet global**;
> - soit un **objet local** à la fonction appelée, objet déclaré ***static***
> (si l'objet n'était pas *static*, il serait détruit lors de la sortie de la fonction appelée
> et la fonction appelante recevrait l'adresse d'un objet qui n'existe plus);
> - soit un **objet dont l'adresse** a elle-même été passée **en paramètre** à la fonction appelée
> (exemples : les fonctions *strcpy()* et *memcpy()* ci-dessus).

Sauf dans le dernier cas, on dit parfois que le pointeur renvoyé par la fonction sert de *poignée* (*"handle"*, en anglais) pour manipuler un objet caché.

Exemple

La fonction *nom_mois()* reçoit en paramètre un numéro de mois; elle retourne à la fonction appelante le nom du mois, plus exactement : un *pointeur* vers ce nom.

Dans la première version ci-dessous, les noms de mois sont mémorisés dans un *tableau de noms* (tableau de caractères à deux dimensions), local à la fonction. Pour que ces noms soient accessibles à la fonction appelante, le tableau doit être déclaré de classe `static`.

```
char * nom_mois (short int mois)
{
   static char nom[12][10] = {"janvier", "février", "mars",
                              "avril", "mai", "juin",
                              "juillet", "août", "septembre",
                              "octobre", "novembre", "décembre"};
/* représentation en mémoire :
    janvier°°°février°°°mars°°°°°°avril°°°°°mai°°°°°°°juin°°°°°°
    juillet°°°août°°°°°°septembre°octobre°°°novembre°°décembre°° */
   return (nom[mois-1]);
}
```

La seconde version ci-dessous utilise un *tableau de pointeurs* vers des chaînes de caractères constantes ... et persistantes; le tableau peut ne pas être déclaré `static` (pour éviter la réinitialisation du tableau à chaque appel de la fonction, il est cependant préférable qu'il le soit).

```
char* nom_mois (short int mois)
{
   char *pt_nom[12] = {"janvier", "février", "mars",
                       "avril", "mai", "juin",
                       "juillet", "août", "septembre",
                       "octobre", "novembre", "décembre"};
/* représentation en mémoire :
    -> -> -> -> -> -> -> -> -> -> -> ->      pointeurs vers :
    janvier°   février°    mars°   avril°      etc.          */
   return (pt_nom[mois-1]);
}
```

4. Les conversions de pointeurs

4.1. Pointeurs génériques

Comment faire pour créer un tableau dont la dimension n'est connue qu'au moment d'exécuter le programme ? Ou pour traiter successivement des tableaux de dimensions variables ? La solution consiste à acquérir dynamiquement un espace contigu de mémoire, d'une taille suffisante. Cet espace est alloué dans une réserve de mémoire appelée "tas" en français, "heap" en anglais.[55] *Pour cela, on invoque la fonction standard* malloc() "memory allocation" *en lui communiquant la taille de l'espace demandé; la fonction retourne, dans un pointeur, l'adresse de début de la zone allouée.*

On peut appeler malloc() *successivement pour obtenir l'espace nécessaire à un tableau de caractères, puis l'espace nécessaire à une variable structurée, puis l'espace nécessaire à un tableau de nombres réels, etc. Le type d'objet repéré par le pointeur de retour diffère d'un appel à l'autre et est inconnu de la fonction* malloc(). *Celle-ci retourne donc un* ***pointeur générique****, c'est-à-dire repérant un objet de type quelconque.*

Un pointeur générique est **déclaré** comme pointant sur le pseudo-type *void* ("vide") : **`void *`** `p` .[56]

Exemple :
```
void * malloc (unsigned int size);
                      /* prototype de malloc() */
```

Utilisation

Un pointeur générique peut contenir l'adresse d'un objet de n'importe quel type. La taille de cet objet étant indéterminée, le déplacement unitaire n'est pas défini pour un tel pointeur. Il s'ensuit qu'* ***on ne peut pas ajouter ou soustraire un déplacement d'adresse à un pointeur générique.*** **Par extension,** ***un pointeur générique ne peut pas non plus servir à un adressage indirect sans déplacement (déplacement 0).***

Si on a déclaré : `void * p_gen;`
on ne peut pas écrire les expressions : `p_gen++` `p_gen-i` `*(p_gen+1)` `*p_gen`

Pour que l'adresse qu'il contient soit utilisable, on doit **convertir** le pointeur générique en pointeur sur un type d'objet déterminé et calculer le déplacement d'adresse par rapport à cet autre pointeur.

La conversion d'un pointeur ne modifie pas l'adresse qu'il contient; elle a pour seul effet de définir le déplacement unitaire applicable.

Une conversion de type impliquant un pointeur générique se fait par n'importe quel mécanisme de ***conversion forcée*** :[57]

- opérateur de *coercition* :
```
void *p;
.....
(float*)p;   /* générique -> sur float */
```

[55] Cf. chapitre 7 : *Organisation de la mémoire allouée à un programme*.
[56] **ATTENTION.** Ne pas confondre **`void *`** `f1()` et **`void`** `f2()` . *f1()* renvoie un pointeur générique, *f2()* ne rend pas de résultat.
[57] Cf. chapitre 4.

- expression d'*affectation* :
```
void* p; long int * p_lint;
.....
p_lint=p;   /* générique -> sur long int */
```
  affectation en *paramètre* d'une fonction
  affectation au *résultat* d'une fonction (par `return`)

Un **test d'égalité** (== ou !=) peut comparer un pointeur générique et tout autre pointeur.

4.2. Pointeurs nuls

Il peut être nécessaire à un programme recourant à l'allocation dynamique de mémoire de savoir si un objet est déjà créé, c'est-à-dire si un espace de mémoire lui est déjà alloué. Le concept de ***pointeur nul*** *est une réponse à ce problème.*

Un pointeur est "nul" lorsqu'il ne repère aucun objet, c'est-à-dire lorsqu'aucune adresse ne lui est affectée. Par convention, un pointeur nul contient la valeur 0.

ATTENTION ! *Un pointeur non initialisé n'est pas "nul" : il ne contient pas 0, mais une adresse inconnue.*

0 est donc une valeur générique admissible pour tout pointeur. Plusieurs fichiers d'en-tête standards, dont <*stdlib.h*>, <*stdio.h*>, la redéfinissent sous le nom mnémonique **NULL** :

```
#define NULL  0
```

La **constante NULL ou 0** peut intervenir dans les **opérations** suivantes :

- *initialisation* d'un pointeur à sa déclaration : `int * p = NULL;`
- opération d'*affectation* : `p = 0;`
  paramètre à l'appel d'une fonction :
  résultat retourné par une fonction : `return NULL;`
- *tests d'égalité* (== !=) : `ptr == 0`

L'existence de pointeurs nuls donne un sens aux **opérations logiques** portant sur des pointeurs :

```
int *p; float *q;          les pointeurs peuvent pointer sur des types d'objets différents
.....
p          vrai si p repère un objet                    faux si p est nul
! p        vrai si p est nul                            faux si p repère un objet
p && q     vrai si  p et q repèrent chacun un objet     faux si l'un des deux est nul
p || q     vrai si p ou q repère un objet               faux si les deux sont nuls
```

4.3. Exemple

La fonction standard *memcpy()* est définie dans le fichier d'en-tête *string.h* de la manière suivante :

```
void *memcpy (void *dest, const void *src, unsigned int n);
```

Elle copie dans le tableau *dest* les *n* premiers octets du tableau *src*, après quoi elle retourne l'adresse du tableau *dest*. Cette fonction suppose que les deux pointeurs *dest* et *src* pointent sur des espaces de mémoire effectivement alloués.

La fonction ci-dessous est une généralisation de *memcpy()*. Si l'espace *dest* n'est pas alloué, on en demande un par la fonction standard d'allocation de mémoire définie comme ceci dans le fichier *stdlib.h* : `void * malloc (unsigned int size);`

```
#include <stdlib.h>        /* déf.  malloc() NULL */
#include <string.h>        /* déf.  memcpy() */
void *copie (void *dest, const void *src, unsigned int n)
/* fonction de copie généralisée
   - obtient un bloc de mémoire de destination si nécessaire
   - en cas d'erreur ou impossibilité, retourne NULL         */
{
            /* N.B.  (ptr==NULL) <=> (!ptr)  */
   if (src==NULL || n <= 0) return NULL;  /* param. invalides */
   if (!dest) dest = malloc(n);        /* allouer un bloc dest */
   if (!dest) return NULL;           /* malloc() n'a pu allouer */
   return memcpy(dest,src,n);     /* copier */
}
```

4.4. Pointeurs sur des objets de types différents

Soit un pointeur ou une adresse repérant un objet de type t_1; par un opérateur de coercition explicite, on peut le convertir en adresse ou pointeur repérant un objet d'un autre type t_2.

Ces conversions n'ont guère de sens qu'entre les membres différents d'une union.

Exemple :

```
struct tete_commande  * pt_tete;
struct ligne_commande  * pt_ligne;
union bon_commande { struct tete_commande tete;
                     struct ligne_commande ligne; }
          en_cours, * pt_bon = &en_cours;
.....
pt_tete = (struct tete_commande  *) pt_bon;
pt_ligne = (struct ligne_commande  *) pt_tete;
```

on peut maintenant écrire :

```
pt_tete->no_client   ⇔   pt_bon->tete.no_client
pt_ligne->qte_commande  ⇔  pt_bon->ligne.qte_commande
```

4.5. Pointeurs et nombres entiers

Un pointeur ou une adresse de mémoire a le même format qu'un ***unsigned int*** (puisque, par définition, le type *int* possède la taille d'un registre adresse). MAIS l'interprétation du contenu est toute différente (que l'on pense ici aux opérations d'addition et soustraction) !

Il est donc possible, sans perte d'information, d'opérer la conversion d'un pointeur ou d'une constante adresse au type *unsigned int* ou à un autre type entier de taille égale ou supérieure; il est aussi possible de convertir une valeur entière en pointeur. La conversion doit être explicite, par le biais d'un opérateur de coercition.

Ce détour permet certains calculs qui ne pourraient être effectués directement sur les pointeurs.

Exemple :

```
struct tete_commande * pt_cde;
int posit_no_client_dans_tete_commande =
   (unsigned)&pt_cde->no_client - (unsigned)pt_cde;
```

5. Synthèse des opérations sur les pointeurs

Nous rappelons ici le *tableau général des opérateurs* du langage C.[58]

Priorité	*Groupes d'opérateurs*	*Opérations*[a]	*	*Notation*	*Associativité*
15	suffixes	**appel de fonction**	*	`x(y,...)`	→
		indexation	*	*`x[y]`*	
		sélection de membre	*	*`x.id`*	
		sélection de membre	*	*`x->id`*	
		post-incrémentation	*	`x++`	
		post-décrémentation	*	`x--`	
14	préfixes	**pré-incrémentation**	*	`++x`	←
		pré-décrémentation	*	`--x`	
		adresse de	*	*`&x`*	
		indirection	*	*`*x`*	
		plus		`+x`	
		moins		`-x`	
		complément sur bits		`~x`	
		négation logique	*	`!x`	
		taille de	*	`sizeof x`	
		conversion au type	*	`(TYPE)x`	
13	multiplicatifs	multiplication		`x*y`	→
		division		`x/y`	
		modulo (reste)		`x%y`	
12	additifs	addition	*	`x+y`	→
		soustraction	*	`x-y`	
11	de décalage	décalage à gauche		`x<<y`	→
		décalage à droite		`x>>y`	
10	relationnels	supérieur	*	`x>y`	→
		supérieur ou égal	*	`x>=y`	
		inférieur	*	`x<y`	
		inférieur ou égal	*	`x<=y`	
09	d'égalité	égal	*	`x==y`	→
		différent	*	`x!=y`	
08	booléens sur bits	ET (intersection)		`x&y`	→
07		OU exclusif		`x^y`	
06		OU inclusif (union)		`x|y`	
05	logiques	ET (produit)	*	`x&&y`	→
04		OU (somme)	*	`x||y`	
03	conditionnel	conditionnel	*	`x?y:z`	←
02	affectation	**affectation absolue**	*	`x=y`	←
		affectation relative	*	`xθ=y`	
01	séquentiel	séquentiel	*	`x,y`	→

* certains opérandes peuvent (ou doivent) être des pointeurs

[a] *en italique*, les opérateurs d'*adressage* en **gras**, les opérateurs effectuant une **affectation**

[58] Cf. chapitre 4.

5.1. Opérateurs d'adressage

Priorité	*Groupes d'opérateurs*	*Opérations*	*Notation*	*Associativité*

15	suffixes	*indexation*
		sélection de membre
		sélection de membre
			`x[y]`
			`x.id`
			`x->id`
				→

14	préfixes	*adresse de* *indirection*	`&x` `*x`	←

- **Manipulation des adresses**

Prise d'adresse : `&x`

L'expression `&x` fournit l'adresse de l'objet *x*. L'opération est évaluée par le compilateur et le résultat est une constante. L'objet *x* ne peut être ni un champ de bits dans une structure, ni une variable de classe *register*, ni le résultat d'une expression (en particulier, d'un appel de fonction).

Adressage indirect : `*x`

L'expression `*x` désigne indirectement l'objet repéré par le pointeur *x*, qui ne peut pas être générique.

Les opérateurs `*` s'associent de droite à gauche : `**x ⇔ *(*x) /* pointeur sur pointeur */`

- **Sélection d'un composant d'une variable agrégat**

Note. Les opérateurs sélectionnant un composant à l'intérieur d'une variable agrégat possèdent la priorité la plus haute. Cette option a pour effet pratique de simplifier l'écriture des références usuelles; sans cette règle, on devrait beaucoup plus souvent employer des parenthèses.

Indexation : `x[y]`

x peut être un élément d'un **tableau de pointeurs**.

Sélection de membre en adressage direct : `x.id`

Pour prendre l'adresse du membre *id* d'une structure ou union *x*, on écrira : `&x.id ⇔ &(x.id)` .

Si le membre *id* est un pointeur, `*x.id ⇔ *(x.id)` effectue un adressage indirect via ce pointeur.

Sélection de membre en adressage indirect : `x->id`

x est un pointeur repérant une structure ou une union. `x->id ⇔ (*x).id` adresse indirectement le membre *id* de la structure ou de l'union repérée par *x*. Pour prendre l'adresse du membre, on écrira : `&x->id ⇔ &(x->id)` .

Si le membre *id* est un pointeur, `*x->id ⇔ *(x->id)` effectue un adressage indirect via ce pointeur.

5.2. Manipulation des types

Priorité	*Groupes d'opérateurs*	*Opérations*	*Notation*	*Associativité*
14	préfixes	taille de conversion au type	`sizeof x` `(TYPE)x`	←

Taille : `sizeof x`

Les expressions suivantes donnent la taille d'une donnée adresse (constante ou pointeur), et non pas la taille de l'objet repéré :

- `sizeof (T *)` où *T* est un nom de type quelconque;
- `sizeof p` où *p* est un pointeur ou une constante adresse.

Conversion de type : `(TYPE)x`

On peut convertir une adresse ou un pointeur *p* quelconque en adresse d'un objet de type *T*, en écrivant : `(T *)p`.

On peut convertir une expression *i* de valeur entière en adresse d'un objet de type *T*, en écrivant : `(T *)i`.

On peut, sans perte d'information, convertir une adresse ou un pointeur *p* quelconque dans le type *unsigned int*, en écrivant : `(unsigned int)p`. La conversion peut également se faire vers un autre type entier.

5.3. Opérations arithmétiques

Priorité	*Groupes d'opérateurs*	*Opérations*	*Notation*	*Associativité*
12	additifs	addition soustraction	`x+y` `x-y`	→

Déplacement d'adresse

A une constante adresse ou un pointeur repérant un élément d'un tableau, on peut ajouter ou soustraire une valeur entière; celle-ci est automatiquement convertie en déplacement d'adresse, en étant multipliée par la taille du type d'élément repéré. Le résultat doit repérer un élément du tableau.

Ces calculs peuvent englober la position suivant immédiatement le dernier élément du tableau.

Compte tenu des priorités,
- `*(p+d)` désigne l'élément d'adresse *p*+*d*
- `*p+d` ⇔ `(*p)+d` augmente de *d* la valeur de l'objet sis à l'adresse *p*

Soustraction de deux pointeurs ou adresses

On peut effectuer une soustraction entre deux pointeurs ou constantes adresses repérant des éléments d'un même tableau. Le résultat est égal à la différence des indices correspondants.

Le calcul peut englober la position suivant immédiatement le dernier élément du tableau.

5.4. Comparaisons

Priorité	*Groupes d'opérateurs*	*Opérations*	*Notation*	*Associativité*

10	relationnels	supérieur
		supérieur ou égal
		inférieur
		inférieur ou égal
			`x>y`
			`x>=y`
			`x<y`
			`x<=y`
				→

09	d'égalité	égal différent	`x==y` `x!=y`	→

Comparaisons dans les limites d'un agrégat

Toutes les comparaisons ont un sens entre deux adresses ou pointeurs quelconques repérant des composants à l'intérieur d'un même agrégat – tableau, union ou structure –, c'est-à-dire dans les limites d'une portion d'espace explicitement contrôlée par une déclaration du programme. (La position suivant immédiatement le dernier élément d'un tableau est assimilée à un de ces éléments.)

Autres comparaisons (tests == !=).

On peut poser un test d'égalité (==) ou d'inégalité (!=) entre les données suivantes :

- entre deux pointeurs ou adresses repérant des *objets de même type*,
  pour savoir s'ils repèrent le même objet;

- entre un pointeur ou une adresse quelconque et un *pointeur générique* `void*` ,
  pour savoir s'ils pointent sur le même objet;

- entre un pointeur et la *constante NULL ou 0*,
  pour savoir si le pointeur repère un objet.

5.5. Opérations logiques

Priorité	*Groupes d'opérateurs*	*Opérations*	*Notation*	*Associativité*
14	préfixes	négation (complément logique)	`!x`	←
05	logiques	ET (produit logique)	`x&&y`	→
04		OU (somme logique)	`x||y`	

Les opérandes *x* et *y* peuvent être des pointeurs quelconques.

Puisqu'il contient 0, un pointeur nul est interprété comme *faux*; un pointeur non nul est interprété comme *vrai*.
Les opérations logiques équivalent à des combinaisons de tests d'égalité/inégalité avec la constante NULL.

5.6. Expression conditionnelle : *expr1* **?** *expr2* **:** *expr3*

• Opération : si *expr1* ≠ 0 (vrai), le résultat est la valeur d'*expr2* (*expr3* n'est pas évalué)
si *expr1* = 0 (faux), le résultat est la valeur d'*expr3* (*expr2* n'est pas évalué)

Chacune des *expr* peut comporter des pointeurs ou des constantes adresses.

Le type du résultat dépend toujours du type des *deux* expr2 *et* expr3 :

– s'il s'agit de deux adresses ou *pointeurs sur des objets de même type*,
le résultat possède le type commun aux deux *expr*;

– s'il s'agit d'un pointeur ou d'une constante adresse et de la *constante NULL ou 0*,
la constante NULL est convertie dans le type de l'autre *expr*;

– s'il s'agit d'un pointeur ou d'une adresse quelconque et d'un *pointeur générique* `void*` ,
le résultat est un pointeur générique `void*` ;

– aucune autre combinaison *expr2/expr3* impliquant des pointeurs n'est autorisée.

Exemple : la fonction suivante retourne un *pointeur* sur une chaîne de caractères.

```
char *prefixe (char sexe, char etat_civil)
{
  return (sexe=='M') ? "Monsieur"
                     : (etat_civil=='C') ? "Mademoiselle"
                                         : "Madame"       ;
}
```

5.7. Opérations d'affectation

Priorité	*Groupes d'opérateurs*	*Opérations*	*Notation*	*Associativité*
02	affectation	**affectation absolue**	`x=y`	←
		affectation relative	`x±=y`	

Affectation absolue : `p = adr`

Si *p* est un pointeur, *adr* peut être un *pointeur générique* ou la constante *générique NULL ou 0*.

De plus,

– si *p* est un *pointeur générique*,
l'évaluation de *adr* doit donner une adresse ou un pointeur quelconque;

– si *p* est un pointeur pour un type d'objet déterminé,
l'évaluation de *adr* doit donner une adresse ou un pointeur valide repérant un objet du *même type*.

Affectation relative : `p += i  p -= i`

`p+=i` est une abréviation de `p=p+i;` `p-=i` est une abréviation de `p=p-i;`

p peut être un pointeur non générique associé à un tableau; alors, *i* doit être une expression de valeur entière, indiquant un déplacement d'adresse valide.

5.8. Incrémentation, décrémentation

Priorité	*Groupes d'opérateurs*	*Opérations*	*Notation*
15	suffixes	post-incrémentation post-décrémentation	`x++` `x--`
14	préfixes	pré-incrémentation pré-décrémentation	`++x` `--x`

L'opérande *x* peut être un pointeur associé à un tableau et dont la mise à jour n'est pas interdite par l'attribut *const*.

L'opération d'incrémentation/décrémentation est souvent associée avec l'indirection. Il importe alors de savoir quelle variable est modifiée : le pointeur ou l'objet sur lequel il pointe ?

Le tableau ci-dessous explique le comportement de ces opérations au moyen de deux équivalences :

- équivalence entre adressage indirect et indexation,
- équivalence entre incrémentation/décrémentation et *expressions séquentielles*.

Si, à l'initialisation, `p = &v[i]`

p affecté	***++p** ⇔ `*(++p)`	`v[++i]`	*i=i+1, v[i]*	*p=p+1, *p*
	***--p** ⇔ `*(--p)`	`v[--i]`	*i=i-1, v[i]*	*p=p-1, *p*
	***p++** ⇔ `*(p++)`	`v[i++]`	*i=i+1, v[i-1]*	*p=p+1, *(p-1)*
	***p--** ⇔ `*(p--)`	`v[i--]`	*i=i-1, v[i+1]*	*p=p-1, *(p+1)*
v affecté	**++*p** ⇔ `++(*p)`	`++v[i]`	*v[i]=v[i]+1, v[i]*	**p=(*p)+1, *p*
	--*p ⇔ `--(*p)`	`--v[i]`	*v[i]=v[i]-1, v[i]*	**p=(*p)-1, *p*
	(*p)++	`v[i]++`	*v[i]=v[i]+1, v[i]-1*	**p=(*p)+1, (*p)-1*
	(*p)--	`v[i]--`	*v[i]=v[i]-1, v[i]+1*	**p=(*p)-1, (*p)+1*

5.9. Appel de fonction : `f(x,...)`

Un appel de fonction peut passer comme paramètre effectif une adresse ou un pointeur; le paramètre formel correspondant est un pointeur permettant à la fonction appelée d'adresser indirectement, dans la fonction appelante, l'objet repéré.

Une fonction peut renvoyer comme résultat une adresse.

L'adresse de la fonction *f* elle-même peut être rangée dans un pointeur.[59]

[59] Cf. chapitre 11.

6. Quelques fonctions utilitaires

6.1. Manipulation des chaînes de caractères (fichier *<string.h>*)

Le fichier d'en-tête standard `<string.h>` définit un certain nombre de fonctions pour la manipulation des tableaux de caractères et chaînes de caractères. Voici les principales.

Les noms de fonctions sont formés des abréviations suivantes : *"CoMPare", "CoPY", "conCATenate", "CHaRacter", "LENgth" – "MEMory", "STRing", "STRing and Number"*.

Dans les définitions qui suivent,

- dans les fonctions *STR[N]...*,
  s1 et *s2* sont des adresses (`char*`) de tableaux ou chaînes de caractères;
- dans les fonctions *MEM...*, *s1* et *s2* sont des pointeurs génériques (`void*`);
- *n* est une valeur entière positive, indiquant une longueur (c'est-à-dire un nombre de caractères);
- *c* est un `int` que la fonction convertit en `char`.

a) Fonctions de comparaison

`int memcmp(s2,s1,n)`	compare les `n` premiers octets de `*s1` et `*s2`
`int strcmp(s2,s1)`	compare les chaînes `*s1` et `*s2` jusqu'à leur terminateur *\0* la chaîne la plus courte est censée prolongée par des caractères *\0*
`int strncmp(s2,s1,n)`	compare les chaînes `*s1` et `*s2` à concurrence de `n` caractères la chaîne la plus courte est censée prolongée par des caractères *\0*

ces fonctions retournent un `int` < 0 si `s2` < `s1` = 0 si `s2` = `s1` > 0 si `s2` > `s1`

b) Fonctions de copie

`*s2` ne peut pas être une constante et doit être de longueur suffisante pour contenir le résultat.
`*s1` et `*s2` doivent occuper en mémoire des espaces disjoints.

`void * memcpy(s2,s1,n)`	copie les `n` premiers octets de `*s1` dans `*s2`
`void * memmove(s2,s1,n)`	*idem*, mais `*s1` et `*s2` peuvent se chevaucher en mémoire
`char * strcpy(s2,s1)`	copie la chaîne `*s1` dans `*s2`, y compris le terminateur *\0*
`char * strncpy(s2,s1,n)`	copie `n` caractères de la chaîne `*s1` dans `*s2` si `*s1` compte moins de `n` caractères, `*s2` est complétée par des *\0*
`char * strcat(s2,s1)`	colle le texte de `*s1` à la suite de celui de `*s2` et termine par *\0*
`char * strncat(s2,s1,n)`	colle au plus `n` caractères de la chaîne `*s1` à la suite du texte de `*s2` et termine par *\0*

chacune de ces fonctions retourne l'adresse `s2` du résultat;
ceci permet d'écrire des expressions comme `txt=strcat(txt,suffix)` analogues à `i=i+1` .

c) Fonctions de recherche

`void * memchr(s2,c,n)`	recherche la première occurrence du caractère `c` dans les `n` premiers octets du bloc `*s2`
`char * strchr(s2,c)`	recherche la première occurrence du caractère `c` dans la chaîne `*s2`
`char * strstr(s2,s1)`	recherche la première occurrence de la chaîne `*s1` (*\0* non compris) dans la chaîne `*s2`

ces fonctions retournent l'adresse de la position trouvée, ou NULL en cas d'échec.

d) Fonctions spéciales

`unsigned int strlen(s1)`	indique le nombre de caractères de `*s1`, terminateur *\0* non compris
`void * memset(s2,c,n)`	place le caractère `c` dans chacune des `n` premières positions de `*s2` `*s2` ne peut pas être une constante renvoie l'adresse `s2`

e) Exemple

Le programme de test ci-dessous demande d'introduire une chaîne de caractères au clavier (la fonction *gets()* accole automatiquement au texte le terminateur *\0*). Le texte introduit est réaffiché sur une ligne complétée par des points jusqu'à la marge de droite.

```
#include <stdio.h>
#include <string.h>

int main(void)
{
  char texte[41], ligne[81];

  puts("tapez un texte [40 caract. max.] :");
  gets(texte);                      /* gets() accole le terminateur \0 */

  memset(ligne,'.',sizeof(ligne)-1);   /* prégarnir la ligne avec ... */
  ligne[sizeof(ligne)-1] = '\0';   /* placer un terminateur de chaîne */
  memcpy(ligne,texte,strlen(texte));   /* copier le texte sans \0 */

  puts(ligne);
  exit(0);        /* ou :   return (0);   = terminaison OK */
}
```

6.2. Allocation dynamique de mémoire (fichier <*stdlib.h*>)

Le fichier standard d'en-tête `<stdlib.h>` contient, entre autres choses, la définition d'un ensemble de fonctions utilitaires pour l'**allocation dynamique de mémoire**. Les blocs de mémoire sont pris dans la réserve que constitue le "tas" ("heap", en anglais).[60]

[60] Cf. chapitre 7.

`void * malloc (size_t t)`	alloue un bloc de mémoire de `t` octets contigus; – rend l'adresse du bloc alloué ou NULL en cas d'échec.
`void * calloc (size_t n, size_t t)`	alloue un bloc suffisant pour stocker `n` objets de taille `t` et initialise tous les octets à *\0*; – rend l'adresse du bloc alloué ou NULL en cas d'échec.
`void * realloc (void * b, size_t t)`	`b` est l'adresse d'un bloc préalablement alloué; la taille de ce bloc est ajustée pour être égale à `t`; le contenu des positions conservées n'est pas modifié; – rend l'adresse du bloc alloué ou NULL en cas d'échec.
`void free (void * b)`	`b` est l'adresse d'un bloc préalablement alloué par une des fonctions précédentes; cet espace est libéré (le programme n'y a plus accès).

Le type `size_t` ("size type") est un type entier non signé défini dans le fichier d'en-tête.

L'espace alloué reste accessible, ... *pourvu que le pointeur renvoyé soit conservé* par le programme.

Exemple

La fonction ci-dessous permute en mémoire deux objets d'un même type quelconque (donc de taille quelconque – cette taille est évidemment donnée en paramètre). L'algorithme utilisable est *temp←a; a←b; b←temp;* – où *temp* est un espace de manoeuvre de taille variable, obtenu par allocation dynamique.

Dans la première version, le pointeur *temp* est une variable automatique, non conservée entre les appels successifs de la fonction. A chaque appel, il est donc nécessaire d'allouer un nouveau bloc de mémoire, et on ne doit pas oublier de libérer, chaque fois, l'espace de manoeuvre alloué ... sans quoi on risquerait d'épuiser la réserve de mémoire.

```
#include <stdlib.h>           /* malloc() free() */
#include <string.h>           /* memcpy() */

void permuter (void* pt_sur_a, void *pt_sur_b, unsigned int taille)
{
   void * temp;               /* variable auto */

   temp = malloc(taille);                  /* allouer */
   memcpy(temp,pt_sur_a,taille);          /* temp <- a */
   memcpy(pt_sur_a,pt_sur_b,taille);      /*  a <- b   */
   memcpy(pt_sur_b,temp,taille);          /* b <- temp */
   free(temp);                           /* libérer !!! */
}
```

Dans la seconde forme, le pointeur *temp* est une variable statique, conservée d'un appel à l'autre; l'espace alloué doit être ajusté à chaque appel. Cette forme est préférable à la précédente.

```
#include <stdlib.h>           /* malloc() realloc() */
#include <string.h>           /* memcpy() */

void permuter (void* pt_sur_a, void *pt_sur_b, unsigned int taille)
{
  static void * temp;                      /* description du */
  static unsigned int long_temp = 0;       /*  bloc  alloué  */

  if (long_temp < taille)      /* faut-il ajuster le bloc ? */
  {
    temp = (long_temp==0) ? malloc(taille) : realloc(temp,taille);
    long_temp = taille;
  }
  memcpy(temp,pt_sur_a,taille);            /* temp <- a */
  memcpy(pt_sur_a,pt_sur_b,taille);        /*  a <- b   */
  memcpy(pt_sur_b,temp,taille);            /* b <- temp */
}
```

6.3. Composition/analyse de texte : *sprintf()*, *sscanf()*

La fonction *printf()* assemble des éléments pour composer un texte qu'elle écrit dans le flot de sortie *stdout*, la fonction *scanf()* isole les éléments d'un texte reçu du flot d'entrée *stdin*.[61] Les fonctions *sprintf()* et *sscanf()* étendent ces techniques de composition ou analyse au traitement de textes en mémoire centrale. Ces fonctions sont décrites dans le fichier d'en-tête `<stdio.h>`.

Comme *printf()*, la fonction ***sprintf()*** – *"print formatted string"* assemble les éléments composant un texte; mais, au lieu de transférer le résultat vers un flot de sortie, elle le range dans un tableau en mémoire centrale, sous la forme d'une chaîne de caractères.

Comme *scanf()*, la fonction ***sscanf()*** – *"scan formatted string"* isole les éléments d'un texte; mais, au lieu de prendre le texte dans un flot d'entrée, elle le prend dans une chaîne de caractères déjà présente en mémoire centrale.

Ces deux fonctions se paramètrent de la même manière que *printf()* et *scanf()*, à ceci près qu'en tête de la liste des paramètres est ajouté un pointeur vers la position courante dans la chaîne de caractères :

```
int sprintf (const char* texte, const char format[], ...);
int sscanf (const char* texte, const char format[], ...);
```

Exemples

```
#include <stdio.h>                                  /* sprintf() */
char * date_txt (struct {short aa, mm, jj;} date)
/* transposer une date numérique dans un format textuel */
{
   static char txt[21];
   char mois [12][4] = {"JAN", "FEV", "MAR", "AVR", "MAI", "JUN",
                        "JUL", "AOU", "SEP", "OCT", "NOV", "DEC"};
   sprintf(txt,"%hd %s 19%02hd",date.jj,mois[date.mm-1],date.aa);
   return (txt);           /* rendre l'adresse du texte (static) */
}
```

[61] Cf. chapitre 3.

Un programme *calculette* doit analyser et exécuter une expression arithmétique introduite au clavier et rangée dans un tableau de caractères *expr*. La fonction *symb_sv()* extrait le symbole suivant de cette expression : nombre, opérateur ou nom de fonction. Elle reçoit en paramètres un pointeur sur la *position* courante dans le texte et un pointeur vers une structure descriptive du *symbole*; au retour, le pointeur indique la position suivante à analyser.

La description d'un symbole comporte un code de *catégorie*, l'indication de la *longueur* et le *texte* du symbole; celui-ci est soit un *nombre* (converti en `float`), soit un *nom* de fonction (on suppose que les majuscules ont été préalablement converties en minuscules), soit un *opérateur* d'un seul caractère. Dans l'expression, les symboles peuvent être séparés par des espaces ou des caractères de tabulation. Par simplification, on suppose ici que le texte introduit ne contient pas d'erreurs.

symb_sv.c

```
#include <stdio.h>            /* sscanf() */
#include <ctype.h>            /* tests des caractères */

   /* descr. d'un symbole : */
typedef struct { int categ, longueur;
                 union { float nbre; char nom[32], oper; } texte;
               } Symbole;
char * symb_sv(char *posit_expr, Symbole *pt_symb)
    /* extraire et décrire le symbole suivant d'une expression,
       à partir de la position courante posit_expr;
       + retourner la position suivante */
{
   while (isspace(*posit_expr)) ++posit_expr;
            /* passer outre des espacements */
   if (*posit_expr == '\0')   /* fin de l'expression : */
   {              /* décrire un symbole fictif conventionnel */
      pt_symb->categ = '0';
      pt_symb->texte.oper = '\0'; pt_symb->longueur = 0;
   }
   else
   if (isdigit(*posit_expr))  /* symb. commençant par un chiffre */
   {              /* extraire un nombre : */
      pt_symb->categ = '9';
      sscanf(posit_expr,"%f%n",
             &pt_symb->texte.nbre,&pt_symb->longueur);
   }
   else
   if (isalpha(*posit_expr))  /* symb. commençant par une lettre */
   {              /* extraire un nom : */
      pt_symb->categ = 'A';
      sscanf(posit_expr,
             "%[abcdefghijklmnopqrstuvwxyz_0123456789]%n",
             &pt_symb->texte.nom,&pt_symb->longueur);
   }
   else                       /* autre caractère */
   {              /* extraire un opérateur : */
      pt_symb->categ = 'X';
      sscanf(posit_expr,"%c%n",
             &pt_symb->texte.oper,&pt_symb->longueur);
   }
   return posit_expr += pt_symb->longueur;  /* position suivante */
}
```

Remarque. A noter, dans l'appel à *sscanf()*, les codes de format suivants :

`%n` nombre de caractères pris dans l'expression => longueur du symbole

`%[abcdefghijklmnopqrstuvwxyz_0123456789]` caract. permis dans un nom

test.c

```
#include "symb_sv.c"

int main(void)          /* test de la fonction symb_sv() */
{
 char expr[80+1], *posit;     /* posit -> position courante */
 Symbole symb;                /* descr. de symbole */
 puts("Introduisez une expression correcte :");
 posit=gets(expr);      /* lire l'expr et initialiser *posit */
 if (!posit) return (1);      /* pas d'expression */
 while (*posit!='\0')   /* jusqu'au bout de l'expression : */
 {
  posit = symb_sv(posit,&symb);            /* extraire */
  switch (symb.categ)                      /* afficher */
  {
   case '9' : printf("\n|%2d|%g",symb.longueur,symb.texte.nbre);
              break;
   case 'A' : printf("\n|%2d|%s",symb.longueur,symb.texte.nom);
              break;
   default  : printf("\n|%2d|%c",symb.longueur,symb.texte.oper);
  }
 }
}
```

Remarque

Entrer un texte en mémoire par la fonction *gets()* puis le découper au moyen de la fonction *sscanf()* présente sur la lecture directe par *scanf()* un avantage appréciable : cela permet de revenir en arrière sur des positions du texte déjà examinées.

6.4. Conversion de nombres (fichier <*stdlib.h*>)

Le fichier d'en-tête standard `<stdlib.h>` propose plusieurs fonctions, moins générales que *sscanf()*, pour convertir une chaîne de caractères représentant un nombre. Voici les plus usitées :

```
int atoi (const char* s)      /* "ASCII string to int" */
long atol (const char* s)     /* "ASCII string to long int" */
double atof (const char* s)   /* "ASCII string to floating pt" */
```

A l'intérieur de la chaîne de caractères examinée, le nombre doit être écrit dans un des formats habituels, tels que ceux que reconnaissent les fonctions de la famille *scanf()*. Les fonctions de conversion se comportent de la même manière que *scanf()* : avancement au delà des caractères d'espacement figurant en tête du texte, conversion, arrêt à la rencontre du premier caractère non convertible.[62]

En cas d'erreur, la variable globale `errno`, définie dans le fichier d'en-tête `<errno.h>`, contient la valeur `ERANGE` ("range error").

[62] En réalité, les fonctions de la famille *scanf()* utilisent elles-mêmes les fonctions de conversion.

6.5 Interaction avec le langage de commande de l'ordinateur (fichier <*stdlib.h*>)

Divers moyens existent pour échanger de l'information entre le langage de commande de l'ordinateur et les programmes dont il gère l'exécution.

• La fonction *main()* d'un programme peut recevoir des **paramètres** de la demande d'exécution de ce programme. *Voir chapitre 11, § 1.2.*

• La fonction `void exit (int code)` est appelée pour **terminer** l'exécution d'un programme et renvoyer au langage de commande un code numérique décrivant la manière dont le programme s'est déroulé. *Voir chapitre 5, § 5.*

• La fonction `system (const char* commande)` demande l'exécution d'une **commande** externe, après quoi le programme appelant se poursuit.

ex.:
```
#include <stdlib.h>
#include <stdio.h>
.....
char repertoire [81];
char commande [121];
.....
printf("dans quel répertoire se trouvent vos fichiers ? ");
scanf(repertoire);
sprintf(commande,"cd %s",repertoire);  /* système MS-DOS : */
system (commande);           /* "change default directory" */
freopen("donnees.dat","r",stdin);/* choix fichier d'entrée */
.....                             /* poursuite du programme */
```

• La plupart des langages de commande permettent de définir des "**variables d'environnement**" contenant du texte quelconque. La fonction `char* getenv (const char* nom_var)` rend un pointeur sur la chaîne de caractères que forme le texte contenu dans la variable dont le nom est donné en paramètre; si aucune variable d'environnement n'est définie sous ce nom, la fonction rend l'adresse `NULL`.

ex.:
```
#include <stdlib.h>
.....
char* repertoire_temporaire = getenv("temp");
..... /* système MS-DOS : la variable "temp" contient
         le nom du répertoire des fichiers temporaires */
```

6.6 Gestion du calendrier et de l'horloge (fichier <*time.h*>)

Le fichier d'en-tête standard `<time.h>` définit des types de données et des fonctions pour la manipulation des dates et heures.

a) Types de données

```
typedef long int  time_t;
  /* secondes écoulées depuis un instant d'origine conventionnel */
```

```
struct tm { int tm_sec,        /* secondes [0..59] */
                tm_min,        /* minutes [0..59] */
                tm_hour,       /* heures [0..23] */
                tm_mday,       /* jour du mois [1..31] !! */
                tm_mon,        /* mois de l'année [0..11] !! */
                tm_year,       /* année du siècle [0..99] */
                tm_wday,       /* jour de semaine (dimanche = 0) */
                tm_yday,       /* jour de l'année [0..365] !! */
                tm_isdst;      /* heure d'été ? 1 = oui, 0 = non,
                                          -1 = info.manquante */
                          };
```

b) Fonctions de calcul

`time_t time (time_t * ptr)` — donne, en secondes, la date et l'heure actuelles (normalement, le paramètre est le pointeur NULL ou 0)

`double difftime (time_t t2, time_t t1)` — donne, en secondes, la différence `t2 - t1`

c) Fonctions de conversion

`struct tm * localtime (const time_t * ptr)` — convertit un temps `time_t` en `tm`

ATTENTION. L'adresse reçue en retour est celle d'une variable interne de la fonction *localtime()*, dont la valeur est mise à jour à chaque appel; il peut donc être nécessaire d'en prendre une copie.

`time_t mktime (struct tm * ptr)` ("make time") — convertit un temps `tm` en `time_t`

d) Fonctions d'affichage

Les fonctions d'affichage des date et heure renvoient l'adresse d'une chaîne de caractères imprimable (car terminée par `\n` "new line"), ayant la forme anglo-saxonne : `Sun Jan  3 15:14:13 1988\n\0`

Cette chaîne de caractères est interne aux fonctions d'affichage et son contenu est modifié à chaque appel.

`char * ctime (time_t * ptr)` ("convert time") temps donné sous le format `time_t`
`char * asctime (struct tm * ptr)` ("ASCII time") temps donné sous le format `tm`

e) Exemple

La fonction *date_du_jour()* fournit la date du jour dans une structure de la forme *jj,mm,19aa*.

```
#include <time.h>
struct date { short int jour, mois, annee; };
struct date date_du_jour (void)
{
   time_t maintenant = time(0);     /* obtenir date et heure */
   struct tm * date_heure = localtime(&maintenant); /* convertir */
   struct date aujourdhui;          /* préparer la réponse */
   aujourdhui.jour = date_heure->tm_mday;       /* 1 .. 31 */
   aujourdhui.mois = date_heure->tm_mon + 1;    /* 1 .. 12 */
   aujourdhui.annee = 1900 + date_heure->tm_year;     /* 19aa */
   return (aujourdhui);             /* répondre */
}
```

Exercices

Pointeurs et structures

1.

Modifier l'exercice d'**impression d'étiquettes d'adresses** proposé au chapitre 8, de manière telle que les fonctions n'échangent plus des structures, mais des *adresses* de structures.

Pointeurs et tableaux

2.

Programmer certaines des fonctions standards de manipulation de chaînes de caractères définies dans le fichier `<string.h>`, par exemple : *strcat(), strchr(), strstr()*. Les programmes peuvent appeler les autres fonctions de la bibliothèque `<string.h>`.

Pour éviter tout problème lors des tests de programme, donner à ces fonctions un nom en majuscules : *STRCAT(), STRCHR(), STRSTR()* ...

Tester ces fonctions.

3.

Modifier les deux versions, récursive et itérative, de la fonction *chercher()* (**recherche dichotomique** dans un tableau) donnée en exemple au chapitre 5, pour qu'elle reçoive en paramètres et renvoie en résultat des pointeurs plutôt que des indices (renvoyer le pointeur NULL en cas d'échec de la recherche).

Pointeurs génériques et nuls

4.

Programmer la fonction standard *calloc()* définie dans le fichier `<stdlib.h>`. Se servir des autres fonctions standards listées aux derniers paragraphes de ce chapitre. Veiller à traiter tous les cas possibles d'erreurs ou incidents.

Conversions de pointeurs

5.

Expliquez l'effet de la macro-définition *dpl()* définie ci-après. En respectant l'ordre d'évaluation des opérateurs, définissez le résultat de chaque étape intermédiaire. *Ecrivez votre explication avant de tester l'exemple fourni.*

```
#define dpl(type_struct,id_membre) \
       ( (unsigned int)&((type_struct *)0)->id_membre )
/* ex. d'utilisation :
  typedef struct {int numero; char nom[32], prenom[16];} identite;
  printf("%u",dpl(identite,prenom));                              */
```

Chapitre 10. Les fichiers

1. Introduction : le concept de fichier

Il convient de donner au mot **fichier** une définition large : toute ***collection de données disponibles dans l'environnement du programme*** en exécution. Il peut s'agir d'une collection enregistrée sur disque ou disquette magnétique; il peut s'agir aussi d'une suite de données introduites au clavier, affichées à l'écran ou imprimées sur papier; il peut encore s'agir de données circulant sur un réseau de communication ...

La bibliothèque standard **`stdio.h`** – *"standard input-output"* définit un certain nombre de fonctions par lesquelles un programme C peut accéder aux fichiers. Les principales sont décrites ci-après.

Une fonction transférant des données du fichier au programme est une fonction de **lecture**; une fonction effectuant le transfert inverse est une fonction d'**écriture**.

Classification des fichiers d'après la nature de leur support

• Supports adressables

Disques et disquettes magnétiques sont des **supports adressables** : la surface de ces supports est découpée en secteurs numérotés dont l'adresse, c'est-à-dire le numéro, peut être calculée par l'ordinateur et directement atteinte par l'organe périphérique de lecture-écriture. A tout moment, il est donc possible d'accéder (se *"positionner"*) à une partie quelconque du fichier. Il s'ensuit notamment que le programme peut *réaccéder* à toute partie qu'il a déjà traitée.

Autre propriété des supports adressables : sur ces supports, un programme peut faire de la **mise à jour**; on entend par là un mélange d'opérations de lecture et d'opérations d'écriture.

• Supports "défilants" ou séquentiels

Sur les appareils qui les manipulent, le papier imprimé, la bande magnétique ont un sens de défilement et, à tout moment, il n'est possible d'accéder qu'à la partie du fichier actuellement présente sous l'organe de lecture ou d'écriture (la *position courante*). Le $i^{ème}$ accès commence à la position *suivant* immédiatement la dernière position traitée lors de l'accès i–1; pour cette raison, de tels supports sont qualifiés de **séquentiels**.

Les messages envoyés sur un réseau de communication, les touches enfoncées à un clavier ... "défilent" semblablement.

De plus, sur ces supports, un programme peut faire soit des opérations de lecture, soit des opérations d'écriture, pas un mélange des deux. (Dans le cas d'une bande magnétique, un *second programme* peut lire un fichier écrit par un *programme précédent*.)

Classification des fichiers d'après la nature de leur contenu

Pour les fonctions d'accès de la bibliothèque `stdio.h`, le contenu d'un fichier peut être interprété de deux manières :

- soit, tout simplement, comme une suite d'octets au contenu quelconque
  - un tel fichier est qualifié de **binaire**;
- soit comme un **texte** découpé en ***lignes***
  - dans les transferts *entre les fonctions d'accès et le programme appelant*, chaque occurrence du caractère `\n` *"new line"* représente la fin d'une ligne.[63]

2. Connexion et déconnexion d'un fichier

2.1. Ouverture d'un fichier – *fopen()*

Aucune action ne peut être entreprise sur un fichier avant que celui-ci ait été connecté au programme, par la fonction ***fopen()*** qui ouvre entre eux un ***flot*** de données. L'effet global de cette opération est de mettre en place les moyens matériels et logiciels nécessaires à la communication avec le fichier.

```
FILE * fopen (char nom_fichier[], char mode[]);
```

• Le premier paramètre de la fonction *fopen()* est l'adresse d'une chaîne de caractères contenant le ***nom*** par lequel le système d'exploitation désigne le fichier. Ce nom peut prendre n'importe quelle forme reconnue par le système d'exploitation. Exemples valides pour le système MS-DOS : *"C:\APPL\ventes\cdes.dat"*,[64] *"CDES.DAT"* ...

• Le deuxième paramètre est l'adresse d'une courte chaîne de caractères indiquant, de manière codée, le ***mode d'ouverture*** souhaité. Ce code signale les intentions du programme par rapport au fichier :

mention obligatoire (1er caractère) :

	r *(read)*	le programme veut lire des données – le fichier doit préexister;
	w *(write)*	le programme veut créer le fichier – si un fichier du même nom préexiste, il est "écrasé";
	a *(append)*	le programme veut augmenter le fichier en écrivant de nouvelles données – si le fichier n'existe pas, il est créé;

mention additionnelle facultative :

	+	le programme souhaite faire de la "mise à jour", c'est-à-dire à la fois des opérations de lecture et d'écriture; *cette option n'est possible que pour un fichier sur support adressable;*

mention facultative :

	b *(binary)*	interpréter le contenu du fichier comme étant binaire
	sinon	interpréter le contenu du fichier comme étant du texte

---

[63] *Sur l'organe périphérique* (disque, imprimante ...), la fin de ligne peut être indiquée autrement, par exemple par une combinaison des deux caractères `\r\n` *"carriage return"* + *"new line"*. Le fait d'interpréter le contenu d'un fichier comme étant un **texte** peut donc impliquer une transformation (automatique) des données véhiculées (conversion entre les différents modes de représentation de la fin de ligne ...). Au contraire, l'interprétation **binaire** n'entraîne aucune transformation des données véhiculées.

[64] Si ce texte est une constante du programme, il devra être libellé : `"C:\\APPL\\ventes\\cdes.dat"`.

• La fonction *fopen()* définit une ***position courante*** dans le fichier : il s'agit du début du fichier (c'est-à-dire la première position occupée), sauf dans le cas d'une ouverture sous le mode *append*, où il s'agit de la fin du fichier (c'est-à-dire la position après la dernière occupée).

```
ffffffffffffffffffffffff
↑                       ↑
```

• Si la connexion réussit, la fonction *fopen()* crée en mémoire centrale une structure décrivant l'état du flot ouvert, structure utilisée par les fonctions d'accès pour gérer le flot. Entre autres informations, cette structure contient l'indicateur de position courante et les indicateurs d'erreur et de fin de fichier (voir ci-après). Le type de cette structure est défini dans le fichier `<stdio.h>` sous le nom **`FILE`**. La fonction rend en résultat l'adresse de cette structure, ou `NULL` en cas d'échec de la connexion; on dit que la fonction *fopen()* rend un ***pointeur de fichier***.[65]

Exemple

Dans un programme de facturation-livraison, on pourrait trouver ce qui suit :

```
FILE *commandes, *clients, *tarif, *stocks, *factures;
.....
commandes = fopen ("cdes.dat", "rb");
clients = fopen ("clients.dat", "rb");
tarif = fopen ("tarif.dat", "rb");
stocks = fopen ("stocks.dat", "r+b");
factures = fopen ("fact.lis", "w");
```

2.2. Fermeture d'un fichier – *fclose()*

La fonction ***fclose()*** déconnecte un fichier, c'est-à-dire qu'elle libère les ressources matérielles (notamment de mémoire) qui lui étaient allouées.

```
int fclose (FILE * ptr_fichier);
```

La fonction *fclose()* reçoit en paramètre le pointeur de fichier décrivant le flot de données à fermer.

En cas d'échec, elle retourne le signal EOF (valeur négative); sinon, elle retourne zéro.

Remarque

Lorsque l'exécution d'un programme se termine, tous les fichiers qu'il a ouverts sont automatiquement fermés s'ils ne l'ont pas été par un appel de la fonction *fclose()*.

[65] Dans le système UNIX et dans les bibliothèques C standards, le terme *descripteur de fichier* possède une autre signification ... c'est pourquoi nous devons ici éviter de l'employer.

2.3. Exemple

La fonction *f_existe()* teste l'existence du fichier dont elle reçoit le nom. Elle le fait en tentant d'ouvrir le fichier pour lecture.

f_existe.c

```
#include <stdio.h>
int f_existe(char nom_fichier[])
{                        /* teste l'existence d'un fichier */
  FILE* ptr_fichier = fopen(nom_fichier,"rb"); /* ouvrir pr lire */
  return (! fclose(ptr_fichier) );             /* fermer */
 /* fopen réussi ... fclose réussi -> =0 ... f_existe -> VRAI
    fopen manqué ... fclose manqué -> <0 ... f_existe -> FAUX */
}
```

2.4. Les flots standards

Au démarrage du programme, *trois flots de textes sont automatiquement ouverts*. Ces flots ont pour rôle d'assurer la communication avec l'utilisateur responsable du programme.[66] A chacun de ces flots standards est associé un *pointeur de fichier constant*, de nom conventionnel :

```
FILE * const stdin;     /* flot standard d'entrée */
FILE * const stdout;    /* flot standard de sortie */
FILE * const stderr;    /* flot standard des messages d'erreur */
```

Le programmeur ne doit pas rédiger ces déclarations; elles sont implicites et ont une portée globale.

La gestion des flots standards ***stdin*** et ***stdout*** se fait au moyen de fonctions spécifiques décrites au chapitre 3. La gestion du flot ***stderr*** doit utiliser les fonctions générales d'accès à un flot de texte quelconque.

3. Traitement des exceptions

Lorsqu'une fonction agissant sur un fichier est incapable d'effectuer l'opération qui lui est demandée, il se produit une condition d'exception. Deux sortes d'exceptions sont distinguées :

- **fin de fichier** en lecture : à la position courante, le fichier ne contient plus de données;
- **erreur** : toute autre cause de non aboutissement de la fonction.

3.1. Fonctions d'analyse des exceptions – *ferror(), feof()*

La structure FILE décrivant l'état de chaque flot de données contient des indicateurs d'erreur et de fin de fichier. Ces indicateurs sont analysés par les fonctions ***ferror()*** et ***feof()*** :

- `feof(`*ptr_fichier_entree*`)`
  est vrai ($\neq 0$) lorsque la lecture a atteint la fin du fichier ("end of file")
- `ferror(`*ptr_fichier*`)`
  est vrai ($\neq 0$) si la dernière opération tentée sur le flot a échoué pour une autre raison

[66] Cf. chapitre 3.

De plus, la variable globale **`errno`**, définie dans le fichier `<stdio.h>` contient un code identifiant la dernière erreur qui s'est produite sur un flot quelconque. Les valeurs de cette variable sont définies dans le fichier `<errno.h>`.[67]

3.2. Retours exceptionnels – *EOF*, *NULL*

Certaines fonctions, comme *fopen()*, dont le résultat est un pointeur, renvoient le pointeur **NULL** (zéro) en cas d'erreur.

Certaines fonctions, comme *fclose()*, dont le résultat est un entier, renvoient la valeur **EOF** (valeur négative) en cas de fin de fichier ("end of file") ... *ou d'erreur*.

4. Fonctions d'accès à un fichier de texte

Les fonctions d'accès aux flots standards *stdin* et *stdout*, décrites au chapitre 3, sont des cas particuliers de fonctions plus générales permettant d'accéder à un fichier de texte quelconque. Nous présentons brièvement ici ces fonctions générales; pour obtenir un supplément d'information, le lecteur se reportera au chapitre 3.

Le tableau ci-dessous classe ces fonctions *d'après la quantité d'information transférée à chaque appel :*

- certaines fonctions transfèrent un seul *caractère*;
- certaines fonctions transfèrent le texte d'une *chaîne de caractères*;
- certaines fonctions transfèrent une ou plusieurs *données* décrites par un "format";

	caractère	**chaîne**	**données formatées**
lecture	`fgetc()` *get character*	`fgets()` *get string*	`fscanf()` *scan with format*
écriture	`fputc()` *put character*	`fputs()` *put string*	`fprintf()` *print with format*

4.1. Lecture/Ecriture d'un caractère – *fgetc()*, *fputc()*

• La fonction **`fgetc()`** lit le caractère situé à la position courante dans un flot de texte en entrée et modifie en conséquence la position courante. Cette fonction est également disponible sous la forme d'une macro-définition de nom **`getc()`**.

```
int fgetc (FILE * ptr_fichier);
int  getc (FILE * ptr_fichier);
```

paramètre : pointeur du flot visé
retour : le caractère lu – EOF en cas d'erreur ou de fin de fichier

*La fonction **`getchar()`** est équivalente à `fgetc(stdin)` .*

[67] Ces valeurs ne sont malheureusement pas standardisées; elles sont particulières à chaque système d'exploitation.

• La fonction **`fputc()`** écrit un caractère à la position courante dans un flot de texte en sortie et modifie en conséquence la position courante. Cette fonction est également disponible sous la forme d'une macro-définition de nom **`putc()`**.

```
int fputc (int caractere, FILE * ptr_fichier);
int  putc (int caractere, FILE * ptr_fichier);
```

paramètres : le caractère à écrire
le pointeur du flot visé
retour : le caractère écrit – EOF en cas d'erreur

La fonction ***`putchar(c)`*** *est équivalente à* `fputc(c,stdout)` .

4.2. Lecture/Ecriture d'une chaîne de caractères – *fgets(), fputs()*

• La fonction **`fgets()`** lit le texte d'une chaîne de caractères à partir de la position courante dans un flot de texte en entrée et modifie en conséquence la position courante.

```
char* fgets (char texte[], int n, FILE * ptr_fichier);
```

paramètres : l'adresse d'un tableau de caractères qui recevra le texte lu
la dimension n de ce tableau
le pointeur du flot visé
retour : l'adresse du tableau – NULL en cas d'erreur ou de fin de fichier

• Le tableau récepteur contient sous la forme d'une chaîne de caractères (clôturée par *\0*) le texte lu; puisque la dimension du tableau est `n`, la fonction lit *au plus* `n-1` caractères.
• Toutefois, si un caractère `\n` de fin de ligne est rencontré avant que le tableau soit rempli, ce caractère de fin de ligne est copié dans le tableau et la lecture s'arrête; le terminateur *\0* est ajouté.

• La fonction **`fputs()`** écrit le texte d'une chaîne de caractères à la position courante dans un flot de texte en sortie et modifie en conséquence la position courante.

```
int fputs (char texte[], FILE * ptr_fichier);
```

paramètres : l'adresse de la chaîne de caractères contenant le texte à écrire
– le terminateur *\0* n'est pas écrit
le pointeur du flot visé
retour : un nombre positif ou nul – EOF en cas d'erreur

ATTENTION ! La fonction ***puts()*** ajoute elle-même le caractère `\n` de fin de ligne au texte qu'elle écrit dans le flot *stdout* et la fonction ***gets()*** supprime elle-même le caractère `\n` de fin de ligne du texte qu'elle lit dans le flot *stdin*.[68] Les fonctions ***fputs()*** et ***fgets()*** ne font rien de tel : le caractère `\n` est présent à l'intérieur des textes transférés.

[68] Cf. chapitre 3.

4.3. Lecture/Ecriture de données formatées – *fscanf(), fprintf()*

• La fonction **`fscanf()`** lit à partir de la position courante dans un flot de texte en entrée une série de données décrites par leurs formats et modifie en conséquence la position courante.

```
int fscanf (FILE * ptr_fichier, char format[], ...);
```

paramètres :	pointeur du flot visé
	chaîne de caractères décrivant les formats d'affichage
	suivie de 0 à n adresses de variables qui recevront les données lues
retour :	nombre de variables garnies – EOF en cas d'erreur ou de fin de fichier

La fonction ***`scanf(...)`*** *est équivalente à* `fscanf(stdin,...)` .

• La fonction **`fprintf()`** écrit à la position courante dans un flot de texte en sortie une série de données décrites par leurs formats et modifie en conséquence la position courante.

```
int fprintf (FILE * ptr_fichier, char format[], ...);
```

paramètres :	pointeur du flot visé
	chaîne de caractères décrivant les formats d'affichage
	suivie de 0 à n valeurs à afficher
retour :	nombre de caractères écrits

La fonction ***`printf(...)`*** *est équivalente à* `fprintf(stdout,...)` .

5. Fonctions d'accès à un fichier binaire

5.1. Lecture/Ecriture d'objets binaires – *fread(), fwrite()*

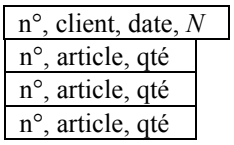
Remarque. Un fichier au *contenu binaire* doit avoir été créé par un programme, avant de pouvoir être lu ou consulté ...

• La fonction ***fwrite()*** écrit dans un fichier binaire *n* objets pris dans un tableau. Ces objets (habituellement, des structures) sont écrits l'un à la suite de l'autre, à partir de la position courante. La position courante est modifiée en conséquence.

```
unsigned int fwrite (void * ptr,
                     unsigned int taille, unsigned int nbre,
                     FILE * ptr_fichier);
```

paramètres :	adresse du tableau d'objets (ou de l'objet unique) à écrire
	taille de chaque occurrence d'objet (élément du tableau)
	nombre d'éléments successifs à extraire du tableau
	pointeur du flot visé
retour :	nombre d'objets écrits avec succès
	– en cas d'erreur, ce nombre est inférieur au nombre passé en paramètre

Exemple. Par un dialogue au terminal, on a introduit en mémoire centrale les informations d'un bon de commande. Ces informations sont représentées de la manière suivante : une structure de type *bon_commande*, formée d'un *entete_commande* (numéro de commande, identité du client, date) suivi d'un tableau *corps* d'un maximum de 15 structures de type *ligne_commande* (numéro de commande, identification de l'article, quantité commandée); l'en-tête comporte un entier indiquant le *nombre de lignes* effectives du bon de commande. Toutes ces informations ayant été validées, on peut les enregistrer dans le fichier des commandes *en cours*.



commande.h

```
      /* types de structures d'un bon de commande : */
typedef struct { short int aaaa, mm, jj; } Date;
typedef struct { int no_commande, no_client;
                 Date date_commande;
                 int nbre_lignes; }
       Entete_Commande;
typedef struct { int no_commande, no_article, qte_commande; }
       Ligne_Commande;
typedef struct { Entete_Commande entete;
                 Ligne_Commande corps [15]; }
       Bon_Commande;
```

```
#include <stdio.h>
#include "commande.h"          /* déf. d'une commande */

      /* fonction externe : */
extern void preparer_commande (Bon_Commande * pt_commande);

void ecrire_commande (Bon_Commande *pt_commande, FILE *pt_fichier)
{
   fwrite (&pt_commande->entete, sizeof(Entete_Commande),
           1, pt_fichier);
   fwrite (&pt_commande->corps, sizeof(Ligne_Commande),
           pt_commande->entete.nbre_lignes, pt_fichier);
}

int main(void)
{
   Bon_Commande commande;

   FILE * en_cours = fopen ("cdes.dat", "ab");        /* mode "a"
       pour écrire à la suite des commandes déjà en cours */
   while (preparer_commande (&commande),
          commande.entete.nbre_lignes > 0)
                /* convention de fin : une commande de 0 ligne */
      ecrire_commande (&commande, en_cours);
   fclose (en_cours);
   exit (0);                   /* code de terminaison OK */
}
```

• La fonction ***fread()*** lit, à partir de la position courante dans un fichier binaire, le contenu de *n* objets (habituellement, des structures), qu'elle range dans un tableau. La position courante est modifiée en conséquence.

```
unsigned int fread (void * ptr,
                    unsigned int taille, unsigned int nbre,
                    FILE * ptr_fichier);
```

paramètres : adresse du tableau (ou de l'objet unique) récepteur
taille de chaque occurrence d'objet (élément du tableau)
nombre d'éléments successifs à ranger dans le tableau
pointeur du flot visé

retour : nombre d'objets obtenus du fichier
– en cas d'erreur ou de fin de fichier,
ce nombre *peut* être inférieur au nombre demandé
!! appeler ensuite les fonctions *feof()* et *ferror()* pour connaître l'état du flot !!

Exemple. Le programme ci-dessous lit le fichier des commandes créé ci-dessus. Pour traiter chaque commande, il appelle une fonction *traiter_commande()* définie ailleurs.

```
#include <stdio.h>
#include "commande.h"          /* déf. d'une commande */

      /* fonction externe : */
extern void traiter_commande (Bon_Commande * pt_commande);

void lire_commande (Bon_Commande * pt_commande, FILE * pt_fichier)
{
   fread (&pt_commande->entete, sizeof(Entete_Commande),
          1, pt_fichier);
   fread (&pt_commande->corps, sizeof(Ligne_Commande),
          pt_commande->entete.nbre_lignes, pt_fichier);
}

int main(void)
{
   Bon_Commande commande;

   FILE * en_cours = fopen ("cdes.dat", "rb");
   while (lire_commande (&commande, en_cours), ! feof(en_cours))
      traiter_commande (&commande);
   fclose (en_cours);
   exit (0);                   /* code de terminaison OK */
}
```

5.2. Positionnement dans un fichier binaire – *fseek()*, *ftell()*

• La fonction ***fseek()*** modifie la position courante dans un fichier binaire. La prochaine opération de lecture ou écriture s'effectuera à partir de la position ainsi établie.[69]

```
int fseek (FILE * ptr_fichier, long int deplacement, int origine);
```

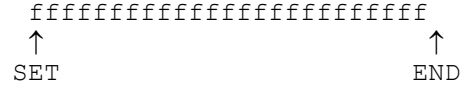
retour : 0 en cas de succès ou $\neq 0$ (pas nécessairement EOF) en cas d'erreur

[69] Exception : si le fichier a été ouvert sous le mode `"a"` ou `"a+"`, toute *écriture* s'effectue à la fin du fichier.

La modification de position s'effectue par addition d'un déplacement par rapport à une origine :

– l'*origine*, indiquée par une valeur codée, peut être une des suivantes[70] :

`SEEK_SET` :	début du fichier (première position occupée = 0),
`SEEK_CUR` :	position courante (position préparée pour le prochain accès),
`SEEK_END` :	fin du fichier (position après la dernière occupée);



– le *déplacement*, compté en octets, est un entier signé
(il doit être ≥ `0` si l'on prend `SEEK_SET` pour origine).

ATTENTION ! En cas d'erreur, la fonction *fseek()* renvoie une valeur non nulle, qui n'est pas nécessairement EOF.

Exemple. Le programme ci-dessous établit une liste des commandes en cours, en imprimant les numéros de commande et de client. Puisque ces données font partie de l'en-tête de la commande, le programme se contente de lire les en-têtes et passe outre du corps de chaque commande.

```
#include <stdio.h>
#include "commande.h"          /* déf. d'une commande */

void main(void)
{
   FILE *commandes, *liste;
   Entete_Commande entete;

   liste = fopen ("cdes.lis", "w");        /* ouvrir la liste */
   commandes = fopen ("cdes.dat", "rb");  /* ouvrir le fichier */
            /* lire un en-tête : */
   while ( fread (&entete, sizeof(Entete_Commande), 1, commandes),
           ! feof(commandes) )             /* fin de fichier ? */
   {        /* traiter l'en-tête lu : */
      fprintf (liste, "commande %05d  -  client %04d\n",
                      entete.no_commande, entete.no_client);
            /* passer outre du corps de la commande : */
      fseek (commandes,
             (sizeof(Ligne_Commande) * entete.nbre_lignes),
             SEEK_CUR);
   }
   fclose (commandes); fclose (liste);    /* fermer les fichiers */
}
```

[70] Un nom `SEEK_xxx` n'identifie pas une variable contenant en valeur la position elle-même, mais un code signifiant que telle position doit être prise comme origine.

• La fonction ***ftell()*** indique, sous la forme d'un `long int`, la position courante dans un fichier. La première position du fichier est la position 0.

```
long int ftell (FILE * ptr_fichier);
```

paramètre : pointeur du flot visé
retour : position courante – EOF en cas d'erreur

Exemple : la fonction *f_taille()* indique la taille d'un fichier déjà ouvert :

```
#include <stdio.h>

long int f_taille (FILE * pt_fichier)   /* taille d'un fichier */
        /* paramètre : pointeur du fichier ouvert */
{
   long int posit_courante, derniere_posit;
   posit_courante = ftell(pt_fichier);          /* mémoriser */
   fseek (pt_fichier,0,SEEK_END);  /* aller à la fin de fichier */
   derniere_posit = ftell(pt_fichier);
   fseek (pt_fichier,posit_courante,SEEK_SET);  /* restaurer */
   return (derniere_posit);        /* répondre */
}
```

5.3. Application : un système de fichiers relatifs

Un **fichier relatif** est un fichier sur support adressable, dont les enregistrements sont numérotés. Il est possible de retrouver immédiatement tout enregistrement, dès lors qu'on en connaît le numéro. Cette organisation implique quelques contraintes :

- tous les enregistrements doivent être de la même longueur;
- la position de l'enregistrement *i* (pour une numérotation commençant à 0) est égale à `SEEK_SET + (sizeof(`*enregistrement*`) * i);`

- si la numérotation n'est pas continue, l'emplacement des enregistrements manquants doit cependant être réservé dans le fichier;
- on doit donc établir une convention pour signaler qu'un emplacement est ou n'est pas occupé.

Nous adopterons pour notre part la convention suivante :

- dans le fichier, tout enregistrement est préfixé par son numéro (de 0 à n–1);
- le numéro d'un emplacement inoccupé sera forcé à la valeur négative EOF.

Le fichier *rel_io.c* définit les fonctions de base pour la gestion d'un tel fichier.

A l'exception des fonctions *ouvrir_rel()* et *fermer_rel()*, ces fonctions reçoivent en paramètres le *numéro* de l'enregistrement à traiter, l'adresse de cet *enregistrement* en mémoire centrale, la *taille* de l'enregistrement, le pointeur du *fichier*. Certaines fonctions ne prennent pas en considération l'adresse de l'enregistrement; dans ce cas, on peut passer l'adresse NULL. Toutes les fonctions renvoient le numéro de l'enregistrement traité ou EOF en cas d'insuccès.

rel_io.c

```
      #define PARAMETRES  int num, void * pt_enreg,\
                          unsigned int taille_enreg, FILE * pt_fichier

      #include <stdio.h>
/*-------------------------------------------------------------------------*/
      #define ouvrir_rel        fopen       /* mode "r+b" ou "w+b" */
      #define fermer_rel        fclose
/*-------------------------------------------------------------------------*/
      int ecrire_rel(PARAMETRES)   /* écrire un enregistrement */
      {
         int taille_totale = sizeof(num) + taille_enreg;

         const static int prefixe = EOF;  /* signal d'emplacement vide */
         int nbre_enreg;          /* nombre d'enregistrements présents */

                  /* déterminer le nombre d'enregistrements présents : */
         for (fseek (pt_fichier,0,SEEK_END),    /* -> fin de fichier */
              nbre_enreg = ftell(pt_fichier)/taille_totale;
               nbre_enreg < num; ++ nbre_enreg) /* tant que nécessaire */
         {        /* créer des enregistrements vides : */
            if (fwrite (&prefixe, sizeof(prefixe), 1, pt_fichier) < 1)
               return (EOF);        /* en cas d'insuccès ... */
            fseek (pt_fichier, taille_enreg, SEEK_CUR);
         }
              /* se positionner sur le préfixe de l'enregistrement : */
         fseek (pt_fichier, (taille_totale * (long)num), SEEK_SET);
              /* écrire le numéro et le texte de l'enregistrement : */
         if (fwrite (&num, sizeof(prefixe), 1, pt_fichier) < 1
           || fwrite (pt_enreg, taille_enreg, 1, pt_fichier) < 1)
            return (EOF);           /* en cas d'insuccès ... */

         return (num);        /* rendre le numéro de l'enreg. écrit */
      }
/*-------------------------------------------------------------------------*/
      int chercher_rel (PARAMETRES) /* vérifier l'existence d'un enreg. :
            la position courante devient celle du préfixe d'enreg. */
         /* l'adresse pt_enreg n'est pas considérée - peut être NULL */
      {
         long int posit;   /* position de l'enreg. dans le fichier */
         int prefixe;      /* réception du numéro lu dans le fichier */
              /* se placer sur le préfixe de l'enreg. et l'extraire : */
         fseek (pt_fichier,
                ((sizeof(prefixe)+taille_enreg) * (long)num), SEEK_SET);
         posit = ftell (pt_fichier);        /* mémoriser la position */
         fread (&prefixe, sizeof(prefixe), 1, pt_fichier);
         if (feof(pt_fichier)) return (EOF);  /* <=> posit. inoccupée */
              /* se replacer au début de l'enregistrement : */
         fseek (pt_fichier, posit, SEEK_SET);

         return (prefixe);          /* numéro ou EOF */
      }
/*-------------------------------------------------------------------------*/
```

```
      int lire_rel (PARAMETRES)   /* lire un enregistrement */
      {
            /* chercher l'enregistrement : */
         int prefixe = chercher_rel (num,NULL,taille_enreg,pt_fichier);
         if (feof(pt_fichier)) return (EOF);   /* <=> posit. inoccupée */

         fseek (pt_fichier, sizeof(num), SEEK_CUR);  /* passer le pfx */
         if (prefixe >=0)     /* si l'emplacement est occupé : */
            fread (pt_enreg, taille_enreg, 1, pt_fichier);   /* lire */
         else                 /* si l'emplacement est inoccupé  */
            fseek (pt_fichier, taille_enreg, SEEK_CUR);     /* passer */

         return (prefixe);          /* numéro ou EOF */
      }
```

On peut définir des fonctions de mise à jour plus sophistiquées, qui vérifient d'abord l'état (le contenu) du fichier et n'effectuent l'opération demandée que si cet état l'autorise. Ces fonctions reçoivent les mêmes paramètres et renvoient le même résultat que les fonctions de base.

- *inserer_rel()* si l'enregistrement n'existe pas, l'écrire;
- *remplacer_rel()* si l'enregistrement existe, écrire sa nouvelle version;
- *supprimer_rel()* si l'enregistrement existe, écrire EOF dans le préfixe;

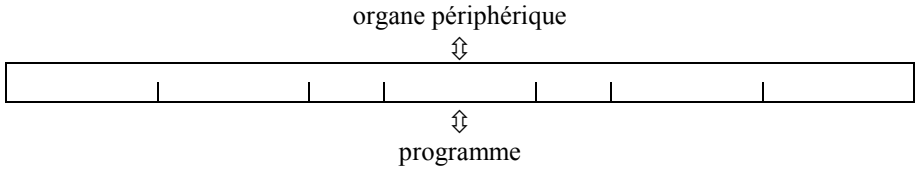
rel_io.c (suite)

```
      int inserer_rel (PARAMETRES)
      {
         if ( chercher_rel(num,NULL,taille_enreg,pt_fichier) != EOF )
            return EOF;
         else return ecrire_rel(num,pt_enreg,taille_enreg,pt_fichier);
      }
/*-----------------------------------------------------------------------*/
      int remplacer_rel (PARAMETRES)
      {
         if ( chercher_rel(num,NULL,taille_enreg,pt_fichier) == EOF )
            return EOF;
         else return ecrire_rel(num,pt_enreg,taille_enreg,pt_fichier);
      }
/*-----------------------------------------------------------------------*/
      int supprimer_rel (PARAMETRES)
        /* l'adresse de l'enreg. n'est pas considérée - peut être NULL */
      {
         if ( chercher_rel(num,NULL,taille_enreg,pt_fichier) == EOF )
            return EOF;
         else
         {
            int prefixe = EOF;    /* marquer l'emplacement inoccupé */
            if (fwrite (&prefixe, sizeof(prefixe), 1, pt_fichier) < 1)
               return EOF;    /* en cas d'insuccès ... */
            fseek (pt_fichier, taille_enreg, SEEK_CUR);     /* passer */
            return num;
         }
      }
```

6. Gestion de la mémoire tampon – fonction *fflush()*

Pour les fonctions d'accès, un fichier est une suite d'octets et chacune de ces fonctions peut transférer un nombre quelconque d'octets contigus. Or tous les octets d'un disque magnétique ne sont pas adressables individuellement; l'unité adressable est le secteur qui contient, par exemple, 512 octets. Chaque accès physique au disque transfère donc un *"bloc"* ou une *"page"* d'un ou plusieurs secteurs. Semblablement, chaque accès physique à une bande magnétique ou à une ligne de télécommunication transfère un bloc d'octets ...

Il est donc nécessaire de créer en mémoire centrale une zone **tampon** intermédiaire ("buffer").



Toute demande d'accès émanant du programme transfère des données entre la mémoire tampon et le programme. Lorsque tout le contenu de la mémoire tampon a été traité, un transfert s'effectue entre le tampon et l'organe périphérique.

• En cas d'écriture, on peut à tout moment appeler la fonction *fflush()* pour forcer le vidage du contenu du tampon, c'est-à-dire son transfert vers l'organe périphérique.[71]

```
int fflush (FILE * ptr_fichier);
```

paramètre : pointeur du flot visé
retour : 0 – EOF en cas d'erreur

Remarque importante sur les opérations de mise à jour

La mise à jour d'un fichier s'effectue par des opérations d'écriture (insertion, remplacement, suppression); habituellement, elle implique également des opérations de lecture (consultation).

La règle suivante est imposée dans l'utilisation des fonctions de la bibliothèque C standard : *entre l'exécution d'une fonction d'écriture et celle d'une fonction de lecture, on doit forcer l'écriture du contenu de la mémoire tampon en appelant la fonction* fflush() *ou une fonction de positionnement dans le fichier.*

On vérifiera aisément que toutes les fonctions d'accès définies ci-dessus dans le fichier *rel_io.c* commencent par effectuer un positionnement dans le fichier, en appelant *fseek()*.

[71] On peut également appeler la fonction *fflush()* pour passer outre du contenu restant dans le tampon d'un fichier en lecture. Par exemple, ayant demandé d'introduire au clavier une réponse d'un caractère, on doit, après avoir lu cette réponse, passer outre des éventuels caractères excédentaires et du terminateur `\n` de fin de réponse; cela peut se faire en programmant : `scanf("%c",&reponse), fflush(stdin);`

Exercices

1.

Récrire l'exercice d'**impression d'étiquettes d'adresses** proposé au chapitre 8, en remplaçant le tableau des fiches signalétiques par un fichier.

Il y a lieu, pour cela, de rédiger deux programmes : le premier crée le fichier signalétique, le second édite les étiquettes.

2.

Ecrire une fonction *f_taille(nom_fichier)* donnant la taille du fichier nommé, ce fichier n'étant pas ouvert. Etablir une convention cohérente pour signaler le cas où le fichier désigné est introuvable.

3.

Ecrire le fichier *rel_io.h* contenant toutes les déclarations nécessaires aux programmes utilisateurs des fonctions du système *rel_io.c* .

4.

Pour tester les fonctions du système *rel_io.c*, réaliser un programme de mise à jour d'un fichier relatif, par exemple : une liste de messages numérotés.

Chapitre 11. Structures de données complexes

Par l'utilisation de pointeurs, il est possible de construire des agencements de données sophistiqués. Ce chapitre a pour ambition d'ouvrir modestement cette perspective.

1. Pointeurs et tableaux

1.1. Tableaux à plusieurs dimensions

Soit un tableau (de nombres `short int`) à deux dimensions :

```
#define L   3
#define C   6
#define T   short int
T t[L][C];
```

Fonctionnellement, c'est-à-dire pour les opérations qui le traitent, il est considéré comme un tableau de 3 lignes et chaque ligne, comme un tableau de 6 colonnes.

```
for (l=0;l<L;++l)
   for (c=0;c<C;++c)
      t[l][c] = (l+1)*(c+1);
```

1	2	3	4	5	6
2	4	6	8	10	12
3	6	9	12	15	18

Matériellement, il est construit dans des positions de mémoire contiguës et prend l'aspect suivant.

	1	2	3	4	5	6	2	4	6	8	10	12	3	6	9	12	15	18
adr	***24***	*26*	*28*	*30*	*32*	*34*	***36***	*38*	*40*	*42*	*44*	*46*	***48***	*50*	*52*	*54*	*56*	*58*

Soit une référence `t[LIGNE][COLONNE]` ;

- le déplacement unitaire pour l'indice `COLONNE` est 2, c'est-à-dire `sizeof(T)`;
- le déplacement unitaire pour l'indice `LIGNE` est 12, c'est-à-dire `sizeof(T)*C`.

L'adresse est égale à `t + ((sizeof(T) * C) * LIGNE) + (sizeof(T) * COLONNE)`
ou `t + (sizeof(T) * ((C * LIGNE) + COLONNE))`.

Pour une référence `t[LIGNE]`, l'adresse est `t + (sizeof(T) * (C * LIGNE))`.

Pour que les calculs d'adresses soient possibles, la dimension C doit être fixe et connue; elle ne peut jamais être indéterminée.

En conséquence, si l'on passe en paramètre à une fonction un tableau à plusieurs dimensions, seule la première dimension peut être déclarée indéterminée. La déclaration du paramètre formel prend la forme **`T t[][C]`** .

Exemple : fonction cherchant un mot dans un tableau (liste) de mots du langage C.

```
#include <string.h>           /* strcmp() */
#define LMAX 31         /* long. maximum d'un mot C */
char * position_mot (char mot[],
                     char liste[][LMAX+1], int nb_mots)
      /* mot[]      : texte cherché : xyz\0
         liste[][x] : tableau des mots enregistrés : xyz\0
         nb_mots    : nombre de mots déjà enregistrés */

   /* N.B. Cet algorithme n'est pas du tout optimisé ! */
{
   int i;         /* indice -> mot */
   for (i=0;i<nb_mots;++i)
      if (strcmp(mot,liste[i])==0) /* si trouvé, renvoyer */
         return (&liste[i]);  /* l'adresse du mot enregistré */
   return (0);          /* renvoie NULL en cas d'échec */
}
```

Adressage par déplacement paramétrable

Une fonction qui, comme la précédente, déclare en paramètre un tableau *`char liste[][LMAX+1]`* , c'est-à-dire sous la forme **`T t[][C]`** , n'est pas tout à fait générale : la longueur maximum des mots est figée. Il vaudrait mieux disposer d'une fonction capable de chercher un mot dans un tableau de mots de longueur maximum quelconque.

Les formules `&t[LIGNE][COLONNE]` ⇔ `t + (sizeof(T) * ((C * LIGNE) + COLONNE))` et `&t[LIGNE]` ⇔ `t + (sizeof(T) * (C * LIGNE))` suggèrent la solution. Il suffit de passer en paramètres les informations suivantes :

- un pointeur **`T *t`** indiquant à la fois l'adresse de début du tableau et le type d'élément,
- la dimension **`C`**.

```
#include <string.h>           /* strcmp() */
char * position_mot (char * mot,
                     char * liste, int l_mot, int nb_mots)
      /* *mot    : texte cherché : xyz\0
         *liste  : tableau des mots enregistrés : xyz\0
         l_mot   : longueur maximum d'un mot dans la liste
         nb_mots : nombre de mots déjà enregistrés */

   /* N.B. Cet algorithme n'est pas du tout optimisé ! */
{
   int i;         /* indice -> mot */
   for (i=0;i<nb_mots;++i)
   {
      char* ptr = liste + (l_mot * i); /* N.B. sizeof(char) = 1 */
      if (strcmp(mot,ptr) == 0)        /* si trouvé, renvoyer */
         return (ptr);         /* l'adresse du mot enregistré */
   }
   return (0);               /* renvoie NULL en cas d'échec */
}
```

1.2. Tableaux de pointeurs

Pour le langage C, un **tableau à plusieurs dimensions** est un tableau de tableaux : une page est un tableau de lignes, une ligne est un tableau de colonnes : `char page[lignes][colonnes];`

Il est souvent plus avantageux d'utiliser un **tableau de pointeurs** : une page est représentée par un tableau de *pointeurs vers* des lignes, chaque ligne est un tableau de colonnes.

Parmi les avantages des tableaux de pointeurs, on notera les suivants.

- **Economie d'espace**

Dans un tableau à deux dimensions `page[lignes][colonnes]`, toutes les lignes – éléments du tableau *page* – ont une taille égale. Le gaspillage d'espace peut être important, si les textes enregistrés ont des longueurs utiles fort différentes ou – cas particulier – si le tableau comporte des cases vides; dans ce cas, un tableau de pointeurs économisera l'espace de mémoire.

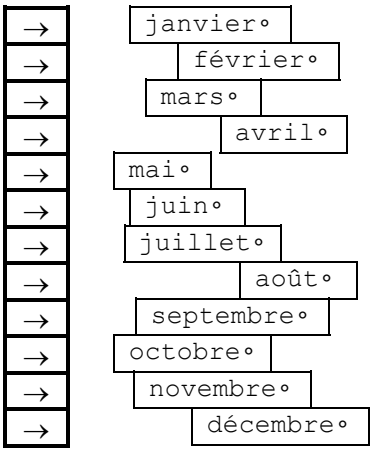
Exemple : *tableau des noms de mois :*

```
char nom_mois[12][10] = { "janvier", "février", "mars",
                          "avril", "mai", "juin",
                          "juillet", "août", "septembre",
                          "octobre", "novembre", décembre"};
```

janvier∘∘∘	février∘∘∘	mars∘∘∘∘∘∘	avril∘∘∘∘∘	mai∘∘∘∘∘∘∘	juin∘∘∘∘∘∘
juillet∘∘∘	août∘∘∘∘∘∘	septembre∘	octobre∘∘∘	novembre∘∘	décembre∘∘

tableau de pointeurs vers les noms de mois :

```
char* nom_mois[12] = { "janvier", "février", "mars",
                       "avril", "mai", "juin",
                       "juillet", "août", "septembre",
                       "octobre", "novembre", décembre"};
```



• **Economie d'opérations**

Supposons une fonction devant trier (c'est-à-dire ordonner) alphabétiquement une série de chaînes de caractères. L'algorithme utilisé est celui du *tri par sélection* :

- sélectionner dans l'intervalle *t[0] ... t[N-1]* le plus petit élément et le permuter avec l'élément *t[0]*;
- sélectionner dans l'intervalle *t[1] ... t[N-1]* le plus petit élément et le permuter avec l'élément *t[1]*;
- continuer de la même manière pour les intervalles *t[2] ... t[N-1]* jusque *t[N-2] ... t[N-1]*.

Si les chaînes de caractères sont stockées dans un tableau de caractères à deux dimensions, les comparaisons et permutations d'éléments devront se faire par les fonctions utilitaires *strcmp()* et *strcpy()*. Si l'on constitue un tableau de pointeurs vers les différentes chaînes, les comparaisons continueront à se faire par la fonction *strcmp()* mais les permutations seront beaucoup plus rapides, puisqu'elles ne déplaceront que des pointeurs et utiliseront les opérateurs de base du langage C.

trier.c

```
#include <string.h>           /* strcmp() */
#define GENERIC void          /* qualificatif de pointeur */

void trier (GENERIC * item[], int nbre_items)
{
   int i, j, min;                          /* indices */
   for (i=0;i<nbre_items-1;++i)
            /* pour les intervalles [0:N-1],[1:N-1],..,[N-2:N-1] */
   {
               /* chercher l'item (chaîne) minimum */
      min = i;
      for (j=i+1;j<nbre_items;++j)
         if (strcmp(item[j],item[min])<0)  min = j;
      if (min != i)
      {
               /* permuter les pointeurs d'items [i] & [min] */
         GENERIC * temp;            /* zone de manoeuvre */
         temp = item[i]; item[i] = item[min]; item[min] = temp;
      }
   }
}
```

Les paramètres de la commande d'exécution

La commande d'exécution d'un programme C peut lui passer des paramètres.

Sur la ligne de commande, le nom du programme est suivi de "mots" séparés par des espaces.

Exemple : `echo  Ah! vous dirais-je, maman ?`
nom du programme : `"echo"`
paramètres (textes) : `"Ah!" "vous" "dirais-je," "maman" "?"`

La fonction ***main()*** reçoit deux paramètres : **`int main (int argc, char *argv[])`**

- un entier `argc` ("argument count") : nombre de chaînes de caractères composant la commande;
- un tableau `argv` ("argument vector") de pointeurs vers les différentes chaînes, *suivis d'un NULL*.

Exemple : `argc : 6  argv [0] →  echo∘`
`[1] →  Ah!∘`
`[2] →  vous∘`
`[3] →  dirais-je,∘`
`[4] →  maman∘`
`[5] →  ?∘`
`[6] NULL`

Le programme standard *echo* copie en sortie ses paramètres, séparés par un espace.

echo.c

```
#include <stdio.h>
int main(int argc, char* argv[])
                    /* argv : tableau de pointeurs */
{
   int i;
   for (i=1;i<argc;++i) /* boucle contrôlée par le compteur argc */
      printf("%s ",argv[i]);
}
```

1.3. Pointeurs de pointeurs

On a vu qu'un paramètre formel déclaré comme tableau peut, de manière équivalente, être déclaré comme pointeur. S'il s'agit d'un tableau de pointeurs (*argv*, dans l'exemple ci-dessus), on peut le déclarer comme un **pointeur (associé à un tableau) de pointeurs**.

Exemples

Le programme echo *s'écrit alors comme ceci :*

echo.c

```
#include <stdio.h>
int main(int argc, char ** argv)
        /* argv : pointeur sur un tableau de pointeurs */
        /* les deux ** marquent le double niveau d'indirection */
{
   while (*++argv)      /* boucle arrêtée par le pointeur NULL */
      printf("%s ",*argv);
}
```

La fonction alloc() *ci-dessous est une fonction unique effectuant toutes les opérations d'allocation de mémoire. Un bloc de mémoire est décrit par deux variables : un pointeur (*ptr*) contenant son adresse et un entier (*long1*) indiquant sa longueur. La fonction* alloc() *reçoit en paramètres ces variables décrivant l'état actuel d'un bloc de mémoire, ainsi qu'un entier (*long2*) indiquant la nouvelle longueur demandée pour le bloc. La combinaison de ces paramètres détermine le comportement de la fonction (voir les commentaires dans le texte du programme). La fonction doit refléter le résultat de son intervention en modifiant les variables descriptives* ptr *et* long1.

alloc.c

```
#include <stdlib.h>     /* malloc(), realloc(), free(), NULL */
#define GENERIC void          /* qualificatif de pointeur */

GENERIC * alloc (GENERIC ** ptr, unsigned int * long1,
                 unsigned int long2)

/* fonction généralisée pour l'allocation dynamique de mémoire */
      /*  **ptr  : adresse du pointeur
          *long1 : adresse de l'indicateur de longueur
          long2  : longueur demandée                    */

{
   if (long2 != *long1)
      /* si (longueur demandée != longueur actuelle),
           modifier l'allocation                        */
   {
      if (long2>0)
         /* si (longueur demandée > 0),
              ajuster (réallouer) si (longueur actuelle != 0)
              allouer si (longueur actuelle == 0)
              modifier l'indicateur de longueur              */
      {
         *ptr = (*long1) ? realloc(*ptr,long2) : malloc(long2);
         *long1 = long2;
      }
      else
         /* si (longueur demandée == 0),
              libérer (désallouer) et annuler le pointeur */
      {
         free(*ptr); *ptr = NULL;
         *long1 = 0;
      }
   }
      /* retourner l'adresse actuelle */
   return (*ptr);
}
```

demo.c

```
#include "alloc.c"

void main(void)
{
   static char *ptr = 0;            /* adresse du bloc alloué  */
   static unsigned int dimens = 0;  /* longueur du bloc alloué */

/*   état initial = {0,0} : rien n'est alloué   */
   alloc(&ptr,&dimens,sizeof"1ère ligne");
/*   état = {->,11} : tableau de 10+1 caractères   */
   alloc(&ptr,&dimens,sizeof"ligne suivante");
/*   état = {->,15} : tableau de 14+1 caractères   */
   alloc(&ptr,&dimens,0);
/*   état = {0,0} : plus rien n'est alloué   */
}
```

2. Pointeurs et structures

2.1. Structures contenant des pointeurs

Une structure peut contenir des pointeurs. Ainsi, dans le dernier exemple ci-dessus, semble-t-il plus naturel de rassembler en une structure unique les deux éléments (adresse et longueur) décrivant un bloc alloué.

alloc.c

```
#include <stdlib.h>     /* malloc(), realloc(), free(), NULL */
#define GENERIC void          /* qualificatif de pointeur */

struct descr_bloc {GENERIC *adr; unsigned int dimens;};
      /* descripteur de bloc */

GENERIC * alloc (struct descr_bloc *p_bloc, unsigned int long2)

/* fonction généralisée pour l'allocation dynamique de mémoire */
/* pour que la fonction puisse modifier le contenu du descripteur,
    elle reçoit un pointeur (*p_bloc) vers le descripteur
        p_bloc-> *adr   : adresse du bloc
                 dimens : longueur du bloc
        long2 : longueur demandée                    */

{
   if (long2 != p_bloc->dimens)
      /* si (longueur demandée != longueur actuelle),
           modifier l'allocation                      */
   {
      if (long2>0)
         /* si (longueur demandée > 0),
              ajuster (réallouer) si (longueur actuelle != 0)
              allouer si (longueur actuelle == 0)
            modifier l'indicateur de longueur              */
      {
         p_bloc->adr = (p_bloc->dimens)? realloc(p_bloc->adr,long2)
                                       : malloc(long2);
         p_bloc->dimens = long2;
      }
      else
         /* si (longueur demandée == 0),
              libérer (désallouer) et annuler le pointeur */
      {
         free(p_bloc->adr); p_bloc->adr = NULL;
         p_bloc->dimens = 0;
      }
   }
      /* retourner l'adresse actuelle du bloc alloué */
   return (p_bloc->adr);
}
```

demo.c

```
#include "alloc.c"

void main(void)
{
   static struct descr_bloc bloc = {0,0};    /* -> bloc alloué  */

/*   état initial = {0,0} : rien n'est alloué   */
   alloc(&bloc,sizeof"1ère ligne");
/*   état = {->,11} : tableau de 10+1 caractères   */
   alloc(&bloc,sizeof"ligne suivante");
/*   état = {->,15} : tableau de 14+1 caractères   */
   alloc(&bloc,0);
/*   état = {0,0} : plus rien n'est alloué   */
}
```

2.2. Listes chaînées

Une structure peut contenir un pointeur repérant un autre objet qui est une structure de même type. Ceci permet de relier ou "chaîner" entre elles deux structures. On crée ainsi ce qu'on appelle une **liste chaînée**. Ce mécanisme ouvre d'énormes possibilités pour la représentation en mémoire centrale de toutes sortes d'ensembles de données. Nous en donnons ci-dessous deux exemples.

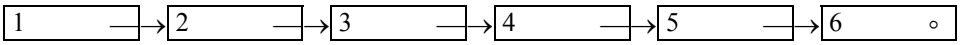
Liste chaînée selon une structure de file

Un fichier de relevés des ventes indique, pour chaque vendeur, son chiffre de vente pour la période concernée; on souhaite connaître le pourcentage des ventes totales que représente le chiffre de vente personnel de chaque vendeur. L'algorithme à mettre en oeuvre est, schématiquement, le suivant :

- *phase 1 : lire les relevés*
  - *pour chaque relevé :*
    - *cumuler son montant dans le montant total*
    - *mémoriser le relevé*
- *phase 2 : relire les relevés mémorisés*
  - *pour chaque relevé :*
    - *calculer le pourcentage*
    - *publier le résultat*

Comment mémoriser temporairement les relevés ? La première idée serait de les ranger dans un tableau en mémoire centrale; mais, le nombre de relevés étant inconnu a priori, il est, en toute rigueur, impossible de déclarer un tableau de dimension suffisante.

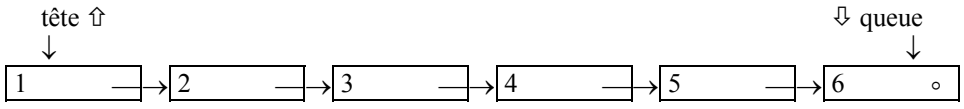
Question générale : comment stocker en mémoire un ensemble d'éléments dans le cas où le nombre d'éléments est inconnu ? En représentant chaque élément par une structure qui, outre les données proprement constitutives de l'élément, contient un *pointeur vers l'élément suivant* (ou précédent) – le dernier élément de la liste contient un pointeur nul. Chaque élément est rangé dans un bloc de mémoire *alloué dynamiquement*; la seule limite est alors la quantité d'espace disponible dans le "tas".



Mémoriser un relevé consiste à insérer un nouvel élément dans la liste chaînée. Relire un relevé consiste à extraire un élément de la liste.

Questions : à quelle position dans la liste doit être inséré un nouvel élément ? de quelle position faut-il extraire l'élément suivant ?

Dans le problème qui nous occupe, on souhaite normalement éditer le résultat dans le même ordre que les données d'entrée. On accèdera donc à la liste chaînée en respectant la logique d'une **file d'attente** (*"queue"*, en anglais) : *premier entré, premier sorti* (cette logique est connue sous le nom de *FIFO – "first in, first out"*). Toute insertion ou entrée dans la file se fait en *queue* de la liste; toute extraction ou sortie de la file se fait en *tête* de la liste. A tout moment, le programme doit donc connaître ces deux positions; elles seront indiquées par des pointeurs; de tels pointeurs sont parfois qualifiés des *pointeurs d'ancrage* (de la liste dans le programme).[72]



- **Structures de données**

La structure représentant un relevé possède le type défini ci-dessous :

```
typedef struct{char nom_vendeur[21]; long int chiffre_vente;}
        Releve;
```

Chaque élément de la liste chaînée aura la structure suivante :

```
typedef struct elemt {struct elemt *suivant; Releve texte;}
        Elemt;
```

`suivant` *est un pointeur vers l'élément suivant dans la liste;*
pour rendre possible la déclaration de son type **`struct`** `elemt` **`*`,**
l'étiquette `elemt` ***doit nécessairement être déclarée avant l'accolade d'ouverture.***

Les pointeurs d'ancrage sont déclarés de la manière suivante :

```
static Elemt *tete = NULL, *queue = NULL;
```

- **Gestion de la file**

Au départ, la liste chaînée est inexistante et les pointeurs d'ancrage sont initialisés à l'adresse NULL

Les fonctions d'insertion et d'extraction d'un élément mémorisé ont la définition suivante :

```
int inserer (Releve * releve), extraire (Releve * releve);
```

- ***paramètre*** *: adresse du relevé courant*
- ***retour*** *:* EOF (< 0) *en cas d'échec,* ≥ 0 *en cas de réussite*

[72] Le concept de ***file*** caractérise le *mode d'accès* aux données, pas leur représentation matérielle. En effet, si l'on connaissait la longueur maximale de la file, on pourrait la matérialiser dans un tableau.

file.c

```
typedef struct {char nom_vendeur[21]; long int chiffre_vente;}
       Releve;
typedef Releve Texte;         /* texte de l'élément */

#include <stdio.h>                  /* EOF */
#include <stdlib.h>                 /* NULL malloc() free() */

typedef struct elemt {struct elemt *suivant; Texte corps;}
       Elemt;           /* type de l'élément chaîné */

static Elemt *tete = NULL, *queue = NULL;       /* ancrages */

int inserer (Texte *ptr_donnees)
   /* insérer un élément en queue de la liste */
{
 Elemt* nouveau = malloc(sizeof(Elemt));  /* allouer une mémoire */
 memcpy(&nouveau->corps,ptr_donnees,sizeof(Texte));    /* copier */
 nouveau->suivant = NULL;          /* -> suivant inexistant */
 if (tete == NULL)      /* si 1er élément ... */
   tete = nouveau;              /* adresse de la tête de liste */
 else                   /* sinon ... */
   queue->suivant = nouveau;   /* chaîner "précédent -> nouveau" */
 queue = nouveau;       /* nouvelle adresse de la queue de liste */
 return (1);
}

int extraire (Texte *ptr_donnees)
   /* extraire un élément en tête de la liste */
{
 Elemt *ancien;                      /* ptr de manoeuvre */
 if (tete == NULL) return (EOF);    /* si liste vide ... EOF */
 memcpy (ptr_donnees,&tete->corps,sizeof(Texte));      /* copier */
 ancien = tete;        /* sauvegarder l'adr. de l'élémt. extrait */
 tete = ancien->suivant;    /* nouvelle adr. de la tête de liste */
 free (ancien);          /* libérer l'espace de l'élémt. extrait */
 return (1);
}
```

- **Application**

ventes.c

```
#include <stdio.h>

       /* type des données : */
typedef struct {char nom_vendeur[21]; long int chiffre_vente;}
       Releve;

   /* fonctions de gestion de la liste mémorisée : */
int inserer (Releve * releve), extraire (Releve * releve);
           /* extraire() renvoie EOF quand la liste est vide */
```

.../...

.../...

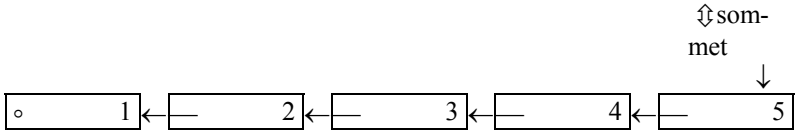
```
void main(void)
{
  Releve releve;        /* relevé en cours de traitement */

      /* ======== PHASE 1 ======== */
  long int total_ventes = 0;
  while
   (scanf("%s %ld",&releve.nom_vendeur,&releve.chiffre_vente)!=EOF)
  {
    total_ventes += releve.chiffre_vente;
    inserer (&releve);                    /* mémoriser */
  }

      /* ======== PHASE 2 ======== */
  while (extraire (&releve) != EOF)       /* relire */
  {
    int pct = (releve.chiffre_vente * 100.0 / total_ventes) + 0.5;
              /* --> int arrondi */
    printf( "%20s   %7ld   %2d %% \n",
            releve.nom_vendeur, releve.chiffre_vente, pct );
  }
}
```

Liste chaînée selon une structure de pile

Une **pile** (*"stack"*, en anglais) est une structure fort semblable à une file. Elle s'en distingue par la logique d'accès : *dernier entré, premier sorti* (*LIFO – "last in, first out"*) : l'insertion et l'extraction d'un élément se font toujours au *sommet* de la pile.[73] Un seul pointeur d'ancrage est nécessaire, qui repère à tout moment ce sommet de la pile. Quant aux éléments de la liste, ils sont chaînés dans l'ordre inverse de celui utilisé dans une file : chaque élément, sauf le premier, pointe vers le précédent.



Les fonctions d'insertion et extraction dans une pile sont connues respectivement sous les noms anglais de *"push"* et *"pop"*.

- **Usage d'une pile**

La relecture d'une pile rend les éléments dans l'ordre inverse de celui de leur introduction.

Tout algorithme récursif se sert d'une pile, implicite, s'il s'agit de la pile des paramètres d'une fonction appelée récursivement, ou explicitement déclarée et gérée par le programme.[74]

---

[73] Le terme *pile* évoque l'analogie de cette logique d'accès avec le mode d'accès à une pile d'assiettes ou une pile de linge dans une armoire ... Le concept de ***pile*** caractérise le *mode d'accès* aux données, pas leur représentation matérielle; en effet, si l'on connaissait la hauteur maximale de la pile, on pourrait la matérialiser dans un tableau.

[74] Cf. chapitre 12.

3. Pointeurs et fonctions

3.1. Pointeurs de fonctions

Revenons à la fonction trier() *applicable à un tableau de pointeurs.*

Dans l'exemple donné plus haut, les pointeurs à ordonner repèrent des chaînes de caractères; la comparaison des éléments se fait donc en appelant la fonction standard strcmp(). *Si chaque pointeur du tableau repérait un objet d'un autre type (un nombre entier, une structure contenant un champ identifiant ...), la fonction* trier() *resterait inchangée, sauf sur le point suivant : une fonction de comparaison spécifique à chaque type d'objet devrait être appelée au lieu de* strcmp().

On peut donc imaginer une fonction trier() *générale qui, outre l'adresse du tableau de pointeurs et sa dimension, recevrait en paramètre l'adresse de la fonction de comparaison à utiliser. Toutes les fonctions de comparaison adopteraient les mêmes conventions que la fonction standard* strcmp() *:*

– *paramètres : deux pointeurs* p1 *et* p2*;*
– *résultat :* `int <0 si *p1<*p2 ==0 si *p1==*p2 >0 si *p1>*p2`

Exemple : comparaison de deux nombres entiers

```
int intcmp (int *p1, int *p2)    /* comparaison de 2 entiers */
{   return (*p1 - *p2);   }
```

Voici des exemples d'appels de la nouvelle fonction trier() *:*

```
char* chaine[100];   /* 100 pointeurs vers des chaînes */
int* nombre[300];    /* 300 pointeurs vers des entiers */
.....
trier(chaîne,100,&strcmp);     ⇔    trier(chaîne,100,strcmp);
trier(nombre,300,&intcmp);     ⇔    trier(nombre,300,intcmp);
```

On voit que le nom d'une fonction désigne en réalité *(une constante contenant) l'adresse de cette fonction* : `strcmp` ⇔ `&strcmp` .[75] Il s'agit de l'adresse de début du corps exécutable `{...}` de la fonction.

Dans la fonction *trier()*, le paramètre formel qui recevra (copie de) cette adresse doit être déclaré comme un **pointeur vers une fonction** :

```
int (* comparer) (void* a, void* b)
```
comparer est un pointeur vers une fonction
qui rend en résultat un entier (`int`).

Les parenthèses autour de **`(`**`*comparer`**`)`** sont obligatoires, car la déclaration suivante a une tout autre signification :

```
int * comparer (void* a, void* b)
```
comparer est une fonction (et non pas un pointeur)
qui rend en résultat un pointeur (`int*`) vers un entier.

[75] On peut faire l'analogie avec le nom d'un tableau, lequel désigne une constante contenant l'adresse de ce tableau.

En effet, l'opérateur `*` d'indirection étant moins prioritaire que l'opérateur `()` d'appel de fonction,

```
        int * comparer(void* a, void* b)
  ⇔     int *(comparer(void* a, void* b))
```

trier.c

```
void trier ( void* item[], int nbre_items,
             int (*comparer)(void* a, void* b) )
                /* comparer : adresse de la f() de comparaison */
{
   int i, j, min;                          /* indices */
   for (i=0;i<nbre_items-1;++i)
            /* pour les intervalles [0:N-1],[1:N-1],..,[N-2:N-1] */
   {           /* chercher l'item minimum */
      min = i;
      for (j=i+1;j<nbre_items;++j)
         if ( comparer(item[j],item[min]) < 0 )  min = j;
      if (min != i)
      {        /* permuter les pointeurs d'items [i] & [min] */
         void * temp;                      /* zone de manoeuvre */
         temp = item[i]; item[i] = item[min]; item[min] = temp;
      }
   }
}
```

Il est possible de définir un tableau de pointeurs de fonctions.

Exemple : au lieu de recevoir en paramètre le nom de la fonction de comparaison, la fonction *trier()* pourrait recevoir un indice sur un tableau de pointeurs de fonctions.

trier.c

```
int strcmp (char*, char*);    /* fonctions  de */
int intcmp (int*, int*);      /*  comparaison  */

void trier ( void* item[], int nbre_items, int mode)
                  /* mode : indice de la f() de comparaison :
                            0->strcmp  1->intcmp       */
{
   static int (*comparer[2])(void* a, void* b) = {strcmp,intcmp};
      /* tableau des fonctions de comparaison */
   int i, j, min;                          /* indices */
   for (i=0;i<nbre_items-1;++i)
            /* pour les intervalles [0:N-1],[1:N-1],..,[N-2:N-1] */
   {           /* chercher l'item minimum */
      min = i;
      for (j=i+1;j<nbre_items;++j)
         if ( comparer[mode](item[j],item[min]) < 0 )  min = j;
      if (min != i)
      {        /* permuter les pointeurs d'items [i] & [min] */
         void * temp;               /* zone de manoeuvre */
         temp = item[i]; item[i] = item[min]; item[min] = temp;
      }
   }
}
```

Syntaxes de déclaration d'un pointeur de fonction

La syntaxe de déclaration d'un pointeur de fonction est assez rébarbative.

Voici, dans le prototype d'une fonction *trier()*, trois manières de déclarer en guise de paramètre un pointeur de fonction.

```
      /* déclaration complète d'un paramètre
         du type pointeur de fonction : */
void trier ( void* item[], int nbre_items,
             int (*comparer)(void* a, void* b) );

      /* déclaration d'un paramètre sans nom
         du type pointeur de fonction : */
void trier ( void**, int,
             int (*)(void*, void*) );

      /* définition préalable d'un type de fonction : */
typedef int comparaison (void* a, void* b);
void trier ( void* item[], int nbre_items,
             comparaison* comparer );
```

Syntaxes d'appel d'une fonction

Soit les déclarations suivantes :

```
      /* constantes fonctions : */
int strcmp (char *a, char *b);      /* extrait de <string.h> */
int intcmp (int *a, int *b);
      /* pointeur de fonction : */
int (*comp) (void *a, void *b) = intcmp;
      /* tableau de pointeurs : */
int (*cpr[2]) (void *a, void *b) = {strcmp,intcmp};
```

Les appels suivants sont tous équivalents :

```
      /* appels par constantes : */
intcmp (&x,&y);               (&intcmp)(&x,&y);
      /* appels par pointeurs : */
comp (&x,&y);                 (*comp)(&x,&y);
cpr[1](&x,&y);                (*cpr[1])(&x,&y);
```

L'opérateur **()** d'appel de fonction étant davantage prioritaire que les opérateurs d'adressage ***** et **&**, les parenthèses sont nécessaires autour des désignateurs `(&intcmp)`, `(*comp)`, `(*cpr[1])`.

Fonctions standards de traitement d'un tableau ordonné : qsort(), bsearch()

Le fichier d'en-tête *<stdlib.h>* fournit, entre autres fonctions, deux algorithmes génériques de tri rapide ("quick sort") et de recherche dichotomique ("binary search") dans un tableau dont les éléments sont de type quelconque. Un des paramètres de ces fonctions est un pointeur vers la fonction de comparaison à employer.

```
void qsort ( void* tableau,
             size_t nbre_elements,
             size_t taille_element,
             int (*comparer) (const void* p1, const void* p2) );

void* bsearch ( const void* cle,   /* fait partie de l'élément cherché */
                const void* tableau,
                size_t nbre_elements,
                size_t taille_element,
                int (*comparer) (const void* cle, const void* element) );
```

La fonction *bsearch()* rend l'adresse de l'élément contenant la "clé" indiquée; elle rend `NULL` si aucun élément ne satisfait cette condition.

La fonction de comparaison doit respecter les conventions de la fonction standard *strcmp()* :

- paramètres : deux pointeurs `p1` et `p2`;
  dans *bsearch()*, `p1==cle` et `p2` désigne successivement les éléments du `tableau`
- résultat : `int` `<0` si `*p1` *avant* `*p2` `==0` si `*p1==*p2` `>0` si `*p1` *après* `*p2`

3.2. Fonctions à liste de paramètres variable

La fonction standard *printf()* et ses dérivées[76] sont des exemples de fonctions recevant des **paramètres en nombre variable et de type variable** : un tableau de caractères puis de 0 à n valeurs de type quelconque.

Déclaration

Une telle fonction est déclarée de la manière suivante :

```
int printf (char * format, ...)
```

- en tête de la liste de paramètres doit figurer *au moins un paramètre nommé* et de type défini (dans l'exemple, le pointeur *format* vers la chaîne de caractères de contrôle);
- en queue de la liste, les points de suspension `...` représentent les derniers paramètres en nombre quelconque (≥ 0) et de type indéterminé.

Appel

Puisque le type des paramètres formels est indéterminé, les paramètres effectifs de l'appel sont convertis en appliquant les règles par défaut :[77]

- un paramètre effectif d'un type entier plus court qu'un `int` subit la *promotion entière*;
- un paramètre effectif de type `float` est converti dans le type `double`;
- un paramètre effectif d'un autre type ne subit aucune conversion.

[76] Cf. chapitre 3, chapitre 10.
[77] Cf. chapitre 4 : *Les appels de fonctions*.

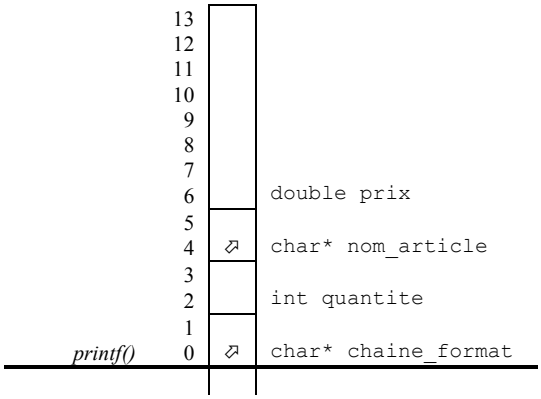
Exemple

Déclarations :
```
/* éléments d'une ligne de facture : */
char nom_article [30+1];
int quantite;
float prix;
.....
```
Appel :
```
printf ("%3d %s : %7.2f", quantite, nom_article, prix);
      /* impression d'une ligne de facture */
```

La fonction *printf()* reçoit, dans l'ordre, les paramètres dans les formats suivants :

adresse :	`char*`	`chaine_format`
nombre entier :	`int`	`quantite`
adresse :	`char*`	`nom_article`
nombre réel :	`double`	`prix`

Lors de l'appel de *printf()*, ces paramètres sont rangés au sommet de la pile :[78]



Exécution

Comment la fonction appelée peut-elle accéder à ses paramètres sans nom ? Par un pointeur qui, successivement, repèrera chacun de ces paramètres. Le fichier d'en-tête standard ***stdarg.h*** – *"standard argument"* – contient les macro-définitions nécessaires :[79]

`va_list`	type du pointeur
`va_start (`*ptr*`, `*dernier_paramètre_nommé*`)`	initialisation du pointeur
`va_arg (`*ptr*`, `*type_du_paramètre*`)`	obtention du paramètre ou argument suivant
`va_end (`*ptr*`)`	clôture

[78] Cf. chapitre 7 : *Organisation de la mémoire allouée au programme*.

[79] Le terme *argument* est souvent employé comme synonyme de *paramètre*. Les noms de macro-définitions commencent par `va_`, abréviation de *"variable argument"*.

• Le pointeur doit être déclaré de type *va_list*.

• *va_start()* doit être appelée avant de traiter le premier paramètre sans nom; elle place dans le pointeur l'adresse du paramètre qui suit le paramètre nommé.

• *va_arg()* renvoie le paramètre repéré par le pointeur puis place le pointeur sur le paramètre suivant; elle est appelée successivement pour chaque paramètre sans nom.

Avant d'appeler *va_arg()*, le programme doit savoir s'il existe encore un paramètre et en connaître le type (type dont la taille sera ajoutée à l'adresse contenue dans le pointeur pour le faire pointer sur le paramètre suivant). *Dans la fonction* printf()*, le paramètre nommé est un texte où chaque code de format* `%` *décrit un des paramètres suivants.*

• *va_end()* effectue le "nettoyage" éventuellement nécessaire; elle doit être appelée avant de retourner à la fonction appelante.

Exemple.[80] Version réduite de *printf()*, ne renvoyant pas de résultat et n'admettant comme codes de formats que `%d`, `%g`, `%s`.

```
#include <stdarg.h>
void printf (char* format, ...)
{
   union { int entier; double reel; char* chaine; } param;
        /* zone de réception du paramètre courant */
   va_list pt_arg;                          /* pointeur d'argument */
   va_start (pt_arg, format);     /* initialisation du ptr */
   for (; *format; ++format)    /*   *format!='\0'   */
   {            /* pour chaque caractère de la chaîne format */
      if (*format!='%')       /* si pas code de format % */
         putchar (*format);         /* imprimer le caractère */
      else                    /* si code de format % */
         switch (*++format)         /* selon le caract. suivant */
         {
            case 'd' : param.entier = va_arg (pt_arg, int);
                       /* traiter le paramètre entier */
                       .....
                       break;
            case 'g' : param.reel = va_arg (pt_arg, double);
                       /* traiter le paramètre réel */
                       .....
                       break;
            case 's' : param.chaine = va_arg (pt_arg, char*);
                       /* traiter le paramètre adresse de chaîne */
                       .....
                       break;
            default  : putchar (*format); /* imprimer le caract. */
         }
   }
   va_end (pt_arg);                 /* clôture */
   return;
}
```

[80] Inspiré d'un exemple de l'ouvrage de Kernighan & Ritchie.

Macro-définitions (fichier <stdarg.h>*)*

Si la pile des paramètres est constituée de la manière illustrée plus haut, les macro-définitions peuvent être les suivantes :

stdarg.h

```
#define va_list                     void*

#define va_start(pt_arg,param1)     (pt_arg = (&param1)+1)
#define va_arg(pt_arg,type)         (*((type*)pt_arg)++)
#define va_end(pt_arg)
```

3.3. Paramètres passés par référence (simulation)

En C, lors d'un appel de fonction, les paramètres sont passés **par valeur** : la fonction appelée reçoit une copie de la valeur de chaque paramètre effectif et travaille sur cette copie. Pour qu'une fonction appelée puisse manipuler un objet *obj* de la fonction appelante, on doit prendre un détour : passer par valeur un pointeur *p_obj* vers l'objet paramètre; la fonction appelée accède alors à l'objet paramètre en **adressage indirect** (`*p_obj`).

Dans d'autres langages de programmation, les paramètres sont passés **par référence** ou **par adresse**, ce qui signifie que la fonction appelée reçoit les adresses des paramètres effectifs; la fonction appelée accède à ces paramètres effectifs en **adressage direct** (`obj`). En C, on peut considérer qu'étant donnée la forme `t[i]` d'une référence indexée, un paramètre tableau est passé par référence; pour les autres paramètres, on peut simuler cet adressage direct en établissant l'équivalence `#define obj (*obj)` , ce qui, au niveau de l'écriture du programme, supprime un niveau d'indirection.

Exemple: fonction permutant les éléments d'une date : *jj,mm,aa* ↔ *aa,mm,jj* .

```
typedef struct {short d[3];} Date_Num;
#define date (*date)  /* paramètre passé par référence (simulé) */
void date_permutee (Date_Num date)
{
  /* modifier le paramètre effectif en adressage direct simulé : */
   short temp = date.d[0];          /* zone de manoeuvre */
   date.d[0] = date.d[2];
   date.d[2] = temp;
}
#undef date
```

texte reçu par le compilateur

```
typedef struct {short d[3];} Date_Num;
void date_permutee (Date_Num (*date))
{
   short temp = (*date).d[0];
   (*date).d[0] = (*date).d[2];
   (*date).d[2] = temp;
}
```

Exercices

Tableaux à plusieurs dimensions – adressage par déplacement paramétrable

1.

Soit deux matrices *M1* et *M2* de dimensions $a \times b$ et $b \times c$ (noter la dimension commune *b*). Le produit de *M1* et *M2* est une matrice *M3* de dimensions $a \times c$, dont chaque élément $m3_{ik}$ est défini comme ceci :

pour $0 \le i < a$ et $0 \le k < c$ $\quad m3_{ik} = \sum_{j=0}^{b-1} m1_{ij} m2_{jk}$

Si l'on représente les trois matrices par trois tableaux *m1[a][b]*, *m2[b][c]* et *m3[a][c]*, l'algorithme du produit de deux matrices est le suivant :

```
for (i=0; i<a; ++i)
   for (k=0; k<c; ++k)
   {
      m3[i][k] = 0;
      for (j=0; j<b; ++j)
         m3[i][k] += m1[i][j] * m2[j][k];
   }
```

Programmer la fonction de calcul du **produit de deux matrices** de dimensions quelconques, dont voici le prototype :

```
float * prod_matr (float *m1, float *m2, int a, int b, int c)
```

La fonction *prod_matr()* doit obtenir l'espace de mémoire nécessaire à la constitution de la matrice produit, espace dont elle renverra l'adresse à la fonction appelante.

Ecrire un programme de test.

Paramètres de la commande d'exécution

2.

Créer un programme *taille* qui affiche la taille des fichiers dont la commande d'exécution donne la liste (ex.: `taille fichier1.dat fichier2.lis fichier3.c`). Signaler les noms qui ne désignent pas des fichiers existants.

Pour obtenir la taille de chaque fichier, appeler la fonction *f_taille()* – exercice 2 du chapitre 10.

Ecrire deux versions de ce programme, l'une utilisant un tableau de pointeurs et l'autre, un pointeur de pointeurs.

Listes chaînées

3.

En s'inspirant du programme *file.c* (cf. page 11-10) de traitement d'une file, créer un programme de traitement d'une **pile**.

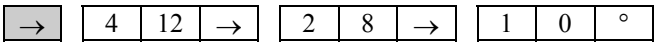
Le programme *vente.c* (cf. page 11-10) peut servir à tester les fonctions de gestion de pile.

4.

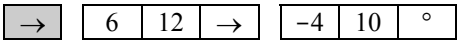
Un **polynôme** $a_{m-1}x^{m-1} + a_{m-2}x^{m-2} \ldots + a_1x^1 + a_0x^0$ est une somme dont chaque terme ou monôme possède la forme ax^e, où x est une variable, a un coefficient non nul, e un exposant entier ≥ 0.

Un tel polynôme peut être matérialisé sous la forme d'une liste chaînée, où chaque monôme est représenté par un noeud contenant un *coefficient* non nul, un *exposant* et un *pointeur* vers le monôme suivant (le pointeur est nul dans le dernier terme).

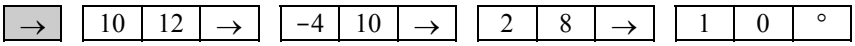
Exemples : $A = 4x^{12} + 2x^8 + 1$



$B = 6x^{12} - 4x^{10}$



$A+B = 10x^{12} - 4x^{10} + 2x^8 + 1$



Mettre au point un programme qui effectue les opérations suivantes :

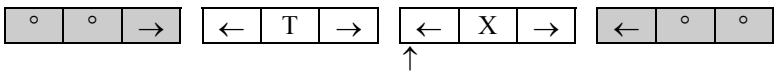
1a) créer deux polynômes sur la base des données introduites au clavier;
1b) à titre de vérification, afficher les deux polynômes ainsi constitués;
2a) créer un polynôme égal à la somme des deux précédents;
2b) afficher le polynôme résultat.

Les procédures (1a) et (2a) consisteront à répéter la séquence suivante :
– déterminer le coefficient et l'exposant du monôme suivant, soit (1a) par lecture au clavier, soit (2a) par calcul ou recopie;
– transmettre ces données en paramètres à une fonction *attacher_terme()*, qui crée un nouveau noeud et en rend l'adresse (à ranger dans le pointeur du noeud précédent ou, pour le premier terme, dans le pointeur d'ancrage).

5.

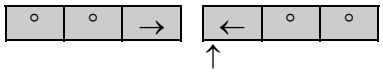
Dans une **liste doublement chaînée**, chaque *maillon* est formé d'un "corps" et de deux pointeurs de chaînage vers le maillon *précédent* et le maillon *suivant*. Ce double chaînage permet de parcourir la liste dans les deux sens avant–arrière.

Pour ancrer la liste dans le programme, au lieu d'employer deux pointeurs de début et fin de liste, il sera plus pratique d'utiliser deux maillons "vides" pointant respectivement vers le premier et le dernier maillons "réels".



pointeur de position courante

Dans une liste vide, ces deux maillons pointent l'un vers l'autre. L'insertion d'un maillon dans une liste vide s'effectue donc de la même façon que dans le cas général : entre deux maillons.



pointeur de position courante

On demande de programmer les fonctions de manipulation d'une liste doublement chaînée, dont chaque maillon a pour "corps" un pointeur `void*` vers un objet de type quelconque. Les fonctions suivantes sont à réaliser :

- initialisation d'une liste : création des maillons d'ancrage;

- déplacement du curseur : modification et renvoi de la position courante :
  - *first(...)* : pointer sur le premier maillon utile,
  - *last(...)* : pointer sur le dernier maillon utile,
  - *forward(current)* : avancer sur le maillon suivant,
  - *backward(current)* : reculer sur le maillon précédent;

- gestion des maillons : (à programmer sans utiliser aucun pointeur de manoeuvre)
  - *insert(current)* : insérer un maillon devant le maillon courant et en renvoyer l'adresse,
  - *remove(current)* : supprimer le maillon courant et placer le curseur sur le suivant;

- accès à l'objet (données) attaché au maillon courant :
  - *attach(current,object)* : attacher l'objet au maillon courant
  - *detach(current)* : détacher l'objet du maillon courant
  - *get(current)* : rendre l'adresse des données

(Toute fonction recevant en paramètre une adresse *current* modifie et renvoie la position courante.)

Ecrire un programme de test, dans lequel les objets attachés aux maillons seront, par exemple, les mots d'un texte.

En guise d'application, réaliser un programme qui simule le **tableau d'affichage des trains au départ** dans une gare :

- une structure `Train` doit être définie (heure prévue, destination, catégorie de train, retard);
- la liste des trains est doublement chaînée;
- on doit pouvoir insérer, supprimer, modifier, déplacer un train dans la liste;
- la fenêtre d'affichage peut se déplacer (avancer ou reculer) dans la liste.

Autre application : un **éditeur de textes** :

- les lignes du texte sont gérées dans une liste doublement chaînée;
- chaque ligne est elle-même une liste de caractères.

6.

Transformer la liste doublement chaînée de l'exercice précédent en **liste circulaire**, dans laquelle les deux maillons vides sont fusionnés en un seul.

7.

Lorsqu'il analyse le texte d'un programme, un compilateur crée en mémoire centrale une **table des symboles**. Chaque fois qu'il rencontre la déclaration d'un symbole (c'est-à-dire un nom), il introduit dans cette table la description du symbole en question (son nom et, sous forme codée, ses attributs). Chaque fois qu'il rencontre une référence à un symbole, il vérifie dans la table que le symbole a été déclaré et qu'il possède les bons attributs.

On demande de créer les fonctions permettant de gérer une table des symboles.

Compte tenu du nombre total de symboles dans le programme (nombre estimé a priori, par exemple en fonction du nombre de lignes du texte), on réserve en mémoire centrale un espace pour un tableau de N descripteurs.

a) Le calcul d'adresse

Pour attribuer un emplacement à un descripteur, on doit faire subir au nom du symbole une double transformation : une fonction *h1()* transforme le nom *symb* en un nombre entier positif *ent* et une fonction *h2()* transforme ce nombre entier en indice *ind* d'emplacement valide, c'est-à-dire qu'elle le ramène dans l'intervalle [0,N[:

$$symb \xrightarrow{h1} ent \xrightarrow{h2} ind$$

Il arrive qu'à partir de deux valeurs de départ V_1 et V_2 différentes, la fonction *h1()* ou la fonction *h2()* donne un résultat identique. On dit qu'alors V_1 et V_2 sont *synonymes* pour la fonction. – Les synonymes demandent un traitement particulier, qui sera décrit au paragraphe suivant.

La fonction *h1()* découpe des "tranches" dans le nom, sur lesquelles elle effectue un calcul donnant un résultat *ent*. Nous adopterons pour cette fonction la formule qui consiste à additionner la valeur numérique binaire de chaque couple de deux lettres consécutives dans le nom. Cette formule donne moins de synonymes que si les tranches découpées comprenaient un seul caractère (car il existe dans le lexique moins de permutations de groupes identiques que de permutations de lettres identiques).

Illustration, en supposant que les majuscules commencent à la valeur numérique 01 :

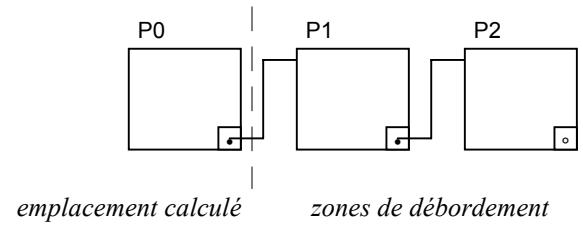
```
  C   03        N   14
+ H   08      + I   09          B   02        C   03
+ I   09      + C   03        + A   01      + A   01
+ E   05      + H   08        + N   14      + P   16
+ N   14      + E   05        + C   03
    = 39          = 39            = 20          = 20

  CH  0308      NI  1409
+ IE  0905    + CH  0308        BA  0201      CA  0301
+ N   1400    + E   0500      + NC  1403    + P   1600
    = 2613        = 2217          = 1604        = 1901
```

La fonction *h2()* consiste à prendre le reste de la division par D : *ind* = *ent* `modulo` *D*, en choisissant comme diviseur le plus grand nombre D inférieur ou égal au nombre N d'emplacements, qui ne possède pas de diviseur (premier) inférieur à 20. REMARQUE : penser au cas particulier de N < 20.

b) Traitement des synonymes

Soit P_0 l'emplacement que le calcul désigne pour stocker un nouveau descripteur. Si cet emplacement est déjà occupé, on lui associe une zone de débordement P_1 en mémoire dynamique, où sera rangé le synonyme de P_0; lorsque P_1 est occupée, on lui associe semblablement une zone P_2 et ainsi de suite. Chaque descripteur P_i contient un pointeur vers le descripteur P_j suivant; le dernier descripteur de la liste contient un pointeur nul.



c) Le programme

On utilisera deux types structurés :

• description d'un `Symbole` : nom (32 caractères, dont le terminateur \0), attributs (suite d'octets codés) et pointeur vers le symbole suivant dans la zone de débordement,

• description de la `Table` : adresse d'implantation en mémoire, nombre d'entrées, diviseur pour la fonction de calcul d'adresse.

Les fonctions suivantes doivent être fournies (chacune peut recourir à des sous-fonctions) :

• `Table creer_table (int nbre_entrees)`
alloue et initialise l'espace de mémoire nécessaire et garnit le descripteur de la table

• `Symbole * ajouter_symbole (Table * p_table, Symbole * p_symbole)`
si le symbole est déjà présent dans la table, la fonction rend un pointeur nul
sinon, elle rend l'adresse de mémoire où elle a copié le descripteur du symbole

• `Symbole * localiser_symbole (Table * p_table, char * nom)`
donne au programme l'adresse du descripteur du symbole dont le nom est fourni en paramètre

Pointeurs de fonctions

8.

En s'inspirant de la fonction *trier()* (cf. page 11-13), programmer une fonction générique de **recherche dichotomique** dans un tableau de pointeurs vers des éléments de type quelconque. Parmi ses paramètres, cette fonction doit recevoir l'adresse de la fonction de comparaison adéquate. Quant à son résultat, il est le pointeur repérant l'élément recherché (et non pas un pointeur vers ce pointeur), ou le pointeur NULL en cas d'échec.

Programmer deux versions de cette fonction, l'une recevant en paramètre un tableau de pointeurs, l'autre recevant des pointeurs de pointeurs.

L'algorithme de la recherche dichotomique est donné à la fin du chapitre 5 (page 5-17).

9.

Programmer une fonction de **tri d'une liste doublement chaînée**. Les éléments de la liste sont de type quelconque. Parmi ses paramètres, la fonction de tri doit recevoir l'adresse de la fonction de comparaison adéquate.

Chapitre 12. Structures récursives

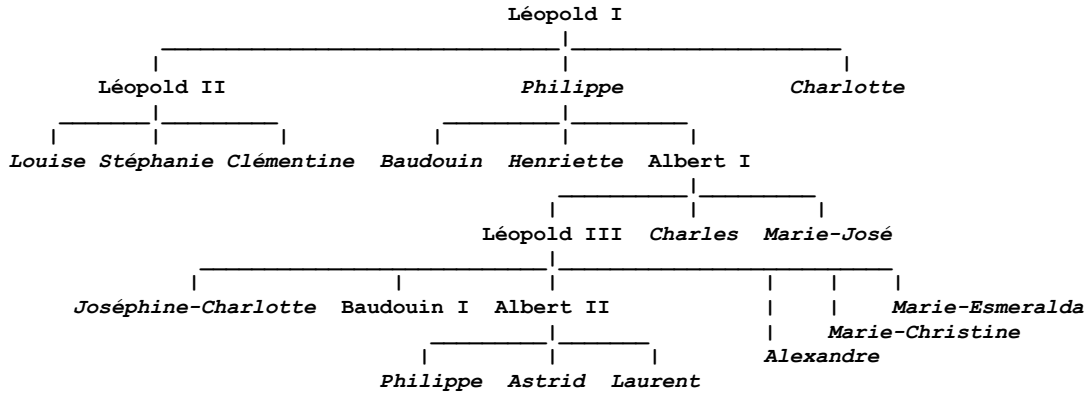
Certains ensembles de données sont définis récursivement. Les algorithmes qui manipulent de tels ensembles sont eux-mêmes toujours récursifs.

1. Parcours d'un arbre de recherche binaire

Le concept d'arbre

La structure d'**arbre** s'illustre facilement par un arbre généalogique.

Exemple : la dynastie belge.



S'il n'est pas *vide*, un **arbre** est un *ensemble* formé d'une ***racine*** (ex.: "Léopold I") associée à $N \geq 0$ (sous-)arbres "fils" *disjoints*. En d'autres termes, chaque fils a un seul père. (Puisqu'un sous-arbre est un arbre, cette définition est récursive.)

Chaque élément d'un (sous-)arbre est un ***noeud***; un noeud sans père est appelé ***racine***; un noeud sans fils est appelé ***feuille*** (ex.: "Clémentine").

Représentation d'un arbre binaire

En mémoire centrale, on représente un arbre de la manière suivante : chaque noeud contient des pointeurs vers ses noeuds fils. Cas particulier : les **arbres binaires**, où chaque noeud contient *deux pointeurs* vers deux noeuds fils; ces pointeurs peuvent être nuls.

Supposons que le type `Texte` décrive les informations utiles placées dans chaque noeud d'un arbre. Chaque noeud d'un arbre binaire possède alors le type suivant :

```
struct noeud {struct noeud *fils_gauche, *fils_droite;
              Texte corps;};
```

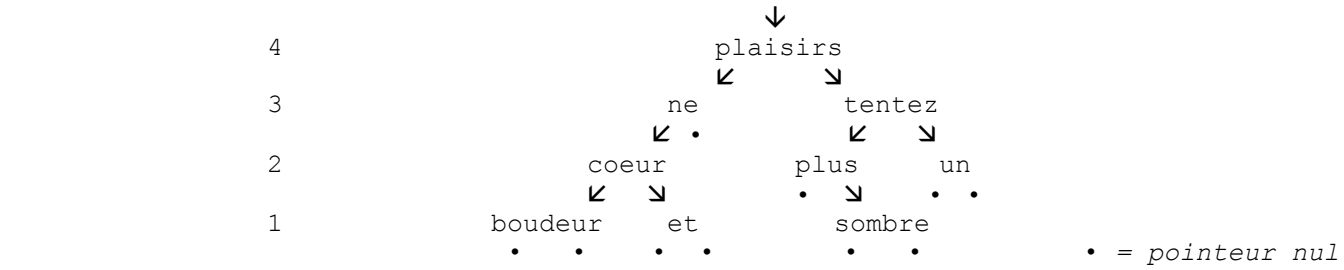
Le pointeur d'ancrage d'un arbre contient l'adresse de la *racine* de l'arbre.

Soit le problème suivant. On veut lister les mots d'un texte dans l'ordre lexicographique (c'est-à-dire alphabétique). La solution consiste à conserver en mémoire tous les mots rencontrés. Chaque mot lu est recherché dans l'ensemble des mots déjà enregistrés; s'il n'existe pas, on le range en mémoire.

Comment organiser l'ensemble des mots en mémoire centrale ? Le premier mot rencontré est placé à la racine d'un arbre binaire. Les mots suivants sont rangés dans les noeuds de l'arbre de manière telle que le sous-arbre gauche de n'importe quel noeud N contienne uniquement des mots classés dans l'alphabet avant le mot du noeud N et que son sous-arbre droit contienne uniquement des mots classés après.

Exemple : arbre binaire construit en lisant le vers ci-dessous :

Plaisirs, ne tentez plus un coeur sombre et boudeur ! (Baudelaire)



Un arbre binaire dont les noeuds sont ordonnés de cette manière est appelé un **arbre binaire de recherche**. Le nombre maximum de noeuds traversés (c'est-à-dire testés) lors d'une recherche est égal à la plus grande hauteur de l'arbre. A la condition que l'arbre soit *équilibré*, c'est-à-dire que toutes ses branches aient la même hauteur, la performance de l'algorithme de recherche dans un arbre binaire est comparable à celle de la recherche dichotomique dans un tableau ordonné : pour un arbre de A noeuds, le nombre maximum de noeuds traversés est de l'ordre de $log_2 A$.

Opérations

Supposons constitué l'arbre ci-dessus.

• ***Lister*** *alphabétiquement* l'arbre de racine "plaisirs" comporte trois phases, dans l'ordre :

- *lister alphabétiquement* le sous-arbre gauche (de racine "ne"),
- *imprimer* le contenu de la racine "plaisirs",
- *lister alphabétiquement* le sous-arbre droit (de racine "tentez").

Cet algorithme est récursif et s'exprime, plus généralement, en ces termes :

• ***Lister*** *alphabétiquement* un (sous-)arbre de racine N :

- *lister alphabétiquement* son sous-arbre gauche s'il existe,
- *imprimer* le contenu de sa racine,
- *lister alphabétiquement* son sous-arbre droit s'il existe.

• ***Montrer*** *la structure* arborescente est un autre algorithme récursif :

> • ***Montrer*** *la structure* d'un (sous-)arbre de racine N :
>
> – *imprimer* le contenu de sa racine,
> – *imprimer* "(",
> – *montrer la structure* du sous-arbre gauche s'il existe,
> – *imprimer* ",",
> – *montrer la structure* du sous-arbre droit s'il existe,
> – *imprimer* ")".

– résultat pour l'exemple ci-dessus : *plaisirs(ne(coeur(boudeur(,),et(,)),),tentez(plus(,sombre(,)),un(,)))*

• ***Insérer*** le mot M dans un arbre de racine N est un algorithme récursif :

– si M = N : le noeud N contient déjà le mot M
– si M < N,
  – si un sous-arbre gauche existe pour N, *insérer* M dans ce sous-arbre
  – sinon, *créer* M comme racine du sous-arbre gauche
– si M > N,
  – si un sous-arbre droit existe pour N, *insérer* M dans ce sous-arbre
  – sinon, *créer* M comme racine du sous-arbre droit

En généralisant, cet algorithme devient :

> • ***Localiser*** le mot M dans un arbre de racine N :
>
> – si M = N : le noeud N contient déjà le mot M
> → effectuer l'action prévue (ex.: lire, remplacer ou supprimer N)
> – si M < N,
> – si un sous-arbre gauche existe pour N, *localiser* M dans ce sous-arbre
> – sinon, le noeud M n'existe pas
> → effectuer l'action prévue (ex.: insérer M comme racine du sous-arbre gauche)
> – si M > N,
> – si un sous-arbre droit existe pour N, *localiser* M dans ce sous-arbre
> – sinon, le noeud M n'existe pas
> → effectuer l'action prévue (ex.: insérer M comme racine du sous-arbre droit)

arbre2.c

```
#include <stdio.h>            /* printf() */
#include <stdlib.h>           /* malloc() */
#include <string.h>           /* strcpy() strcmp() */

typedef struct noeud {struct noeud *fils_g, *fils_d;
                      char texte [15+1];}          Noeud;
/* N.B. pointeur ppn -> pointeur modifiable pn -> noeud n */

static void creer (Noeud** ppn, char* mot)
    /* créer un noeud et placer son adresse
       dans le pointeur (*ppn) du noeud père */
{
   Noeud* pn = *ppn = malloc(sizeof(Noeud));  /* obtenir mémoire */
   pn->fils_g = NULL; pn->fils_d = NULL;      /* chaînages vides */
   strcpy (pn->texte, mot);           /* ACTION : créer le texte */
}
```

```
void localiser (Noeud** ppn, char* mot)
    /* chercher et insérer un noeud ds le ss-arbre d'adr. (*ppn) */
{
   if (*ppn == NULL)          /* si le sous-arbre n'existe pas : */
      creer (ppn, mot);       /* ACTION pour le cas "non trouvé" */
   else
   {
      Noeud* pn = *ppn;        /* supprimer une indirection */
      int test = strcmp (mot, pn->texte);
      if (test == 0)  ;       /* ACTION pour le cas "trouvé" */
      if (test < 0) localiser (&pn->fils_g, mot); /* réc.gauche */
      if (test > 0) localiser (&pn->fils_d, mot); /* réc.droite */
   }
}

void lister (Noeud* pn)             /* liste alphabétique */
{
   if (pn != NULL)            /* si le sous-arbre (*pn) existe : */
   {
      lister (pn->fils_g);                /* récurrence à gauche */
      printf ("%s \n", pn->texte);        /* ACTION */
      lister (pn->fils_d);                /* récurrence à droite */
   }
}

void montrer (Noeud* pn)            /* structure arborescente */
{
   if (pn != NULL)            /* si le sous-arbre (*pn) existe : */
   {
      printf (pn->texte);                 /* ACTION */
      printf ("(");
      montrer (pn->fils_g);               /* récurrence à gauche */
      printf (",");
      montrer (pn->fils_d);               /* récurrence à droite */
      printf (")");
   }
}
```

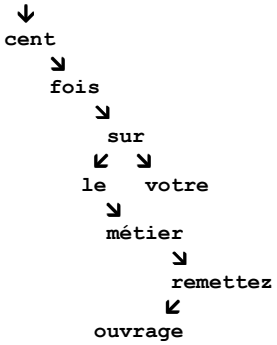
test.c

```
#include "arbre2.c"
int main(void)    /* tests des opérations sur un arbre binaire */
{
   char phrase[]        /* phrase analysée */
      = "plaisirs ne tentez plus un coeur sombre et boudeur ";
   char* posit;         /* position courante dans la phrase */
   int deplact;         /* nombre de caractères lus */
   char mot[15+1];      /* réception du mot courant */
   void* arbre = NULL;  /* ancrage : ptr vers racine de l'arbre */
   for (posit=phrase;
        sscanf(posit,"%s%n",&mot,&deplact) > 0; /* lu un mot ? */
        posit+=deplact)
                  localiser(&arbre,mot);     /* placer le mot */
   lister(arbre);             /* liste alphabétique */
   montrer(arbre);            /* structure arborescente */
}
```

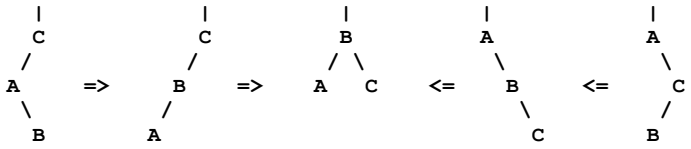
Arbre AVL[81]

Certains textes peuvent donner un arbre binaire fort déséquilibré.

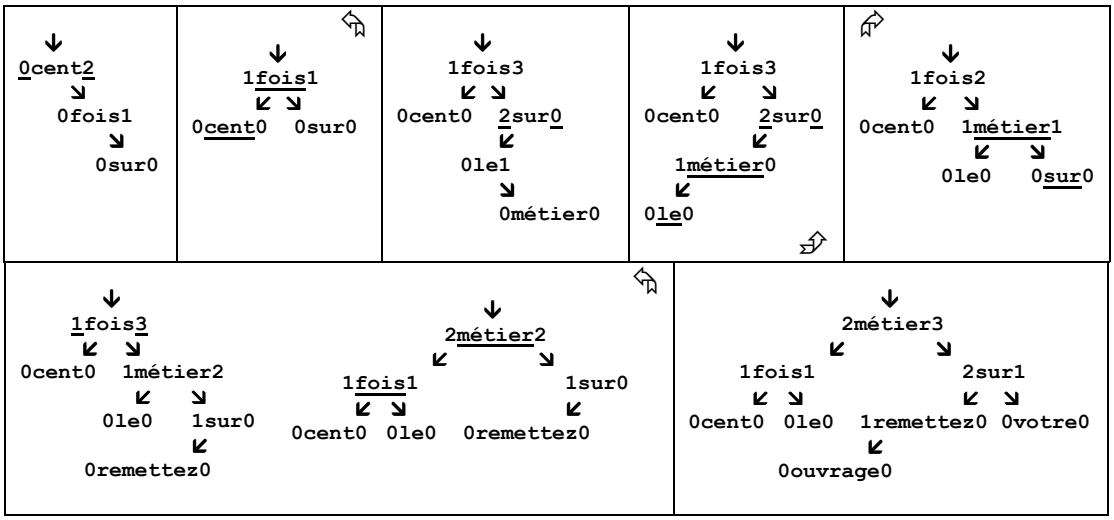
Exemple : *Cent fois sur le métier remettez votre ouvrage.* (Boileau)



Des techniques de rotation sont utilisées pour améliorer l'équilibre d'un tel arbre. Ces techniques sont mises en oeuvre chaque fois que la différence de hauteur entre les sous-arbres gauche et droit d'un noeud dépasse 1. Elles opèrent par des échanges de pointeurs.



Exemple.



La structure d'un noeud doit être adaptée : outre l'adresse de la racine de chacun des deux sous-arbres fils, tout noeud contient une indication de la hauteur de ses deux sous-arbres fils. Les algorithmes utilisés pour l'insertion d'un noeud dans un sous-arbre doivent ajuster dans le noeud père les deux éléments composant la description du sous-arbre modifié.

---

[81] Initiales des auteurs soviétiques ADELSON-VELSKII & LANDIS, qui ont décrit cette structure en 1962.

arbreAVL.c

```
#include <stdlib.h>           /* calloc() */
#include <string.h>           /* strcmp() strcpy() */
#define PRIVATE static  /* fonctions appelées par localiser() */

typedef struct {struct noeud *adr; short int haut;} Arbre;
/* REM.: la définition du descripteur d'arbre fait mention
         de la structure 'noeud', définie ultérieurement     */
typedef struct noeud {Arbre fils [2]; char texte [15+1];} Noeud;
typedef enum {gche = 0, drte = 1} Indice;

/* N.B. pointeur pda -> descripteur modifiable de sous-arbre
                        pointeur modifiable pn -> noeud n */

PRIVATE void creer (Arbre* pda, char* mot)
   /* créer un noeud et placer sa description dans le noeud père */
{
 pda->adr = calloc(1,sizeof(Noeud)); /* obtenir mémoire mise à 0 */
 pda->haut = 1;
 strcpy (pda->adr->texte, mot);       /* ACTION : créer le texte */
}

/**/ PRIVATE void equilibrer (Arbre* pda);

void localiser (Arbre* pda, char* mot)
  /* chercher et insérer un mot ds le ss-arbre d'adr. (pda->adr) */
{
 if (pda->adr == NULL)        /* si le sous-arbre n'existe pas : */
   creer (pda, mot);          /* ACTION pour le cas "non trouvé" */
 else
 {                                  /* chercher et insérer le mot */
  int test = strcmp (mot, pda->adr->texte);
  if (test == 0)  ;               /* ACTION pour le cas "trouvé" */
  if (test < 0) localiser (&pda->adr->fils[gche], mot);
  if (test > 0) localiser (&pda->adr->fils[drte], mot);
  equilibrer (pda);     /* rééquilibrer et déterminer la hauteur */
 }
}

#define max(a,b) ((a)>(b)?(a):(b))           /* maximum de (a,b) */

/**/ PRIVATE void pivoter (Arbre* pda, Indice sens); /* rotation
    ... dans le sens indiqué, du sous-arbre de racine (pda->adr) */

PRIVATE void equilibrer (Arbre* pda)
{
   /* rééquilibrer par rotation : */
 if (pda->adr->fils[gche].haut - pda->adr->fils[drte].haut > 1)
   pivoter (pda, drte);
 else
 if (pda->adr->fils[drte].haut - pda->adr->fils[gche].haut > 1)
   pivoter (pda, gche);
   /* calculer la hauteur de l'arbre : */
 pda->haut = 1 + max (pda->adr->fils[gche].haut,
                      pda->adr->fils[drte].haut);
}
```

```
PRIVATE void pivoter (Arbre* pda, Indice sens)
      /* rotation dans le sens indiqué */
      /*** exemple illustré : rotation à droite ***/
{
 Indice contresens = gche + drte - sens;
 Noeud* pn = pda->adr;
 Noeud* rn;                      /* pointeur  ->  nouvelle racine */
 if (pn->fils[contresens].adr->fils[sens].haut
                          > pn->fils[sens].haut)
            /***   *pda|pn->      2[C]0
                                    /
                                 0[A]1
                                    \
                                   0[B]0                ***/
   pivoter (&pn->fils[contresens], contresens); /*rotation double*/
 pda->adr = rn = pn->fils[contresens].adr;    /* nouvelle racine */
            /***        pn->        2[C]0
                                     /
                     *pda|rn->      1[B]0
                                     /
                                 0[A]0                ***/
 pn->fils[contresens] = rn->fils[sens];
 rn->fils[sens].adr = pn;
 rn->fils[sens].haut = 1+max(rn->fils[sens].adr->fils[gche].haut,
                             rn->fils[sens].adr->fils[drte].haut);
            /***   *pda|rn->      1[B]1
                                  /   \
                              0[A]0   0[C]0  <-pn     ***/
}
```

Commentaire

Les fonctions de parcours d'un arbre – *localiser(), lister(), montrer()* – sont **récursives** : elles s'appellent elles-mêmes. La fonction de rotation *pivoter()* est également récursive : dans certains cas, la rotation à gauche appelle la rotation à droite, et réciproquement.

De la même manière que les algorithmes de tri ne dépendent pas du type des objets à trier,[82] les algorithmes de parcours d'un arbre demeurent inchangés quel que soit le type des objets énumérés. Il est dès lors intéressant de les réaliser sous la forme de **fonctions génériques**. Pour cela,

– le corps d'un noeud ne doit pas être formé du texte de l'objet, mais d'un pointeur générique vers cet objet;
– chaque fonction de parcours doit recevoir en paramètres les adresses des fonctions de comparaison et d'"action" qui manipulent les objets localisés, ces fonctions n'effectuant aucune manipulation sur les pointeurs de chaînage.

[82] Cf. chapitre 11.

2. Règles de programmation des fonctions récursives

2.1. Terminaison de l'algorithme

Pour que l'exécution d'un algorithme récursif ne se répète pas à l'infini, ***tout appel récursif doit être conditionnel***, c'est-à-dire rédigé à l'intérieur d'une *construction alternative* (`if` ou `switch`) dans laquelle il existe une issue non récursive.

> *Exemple.* Les deux appels récursifs de la fonction *localiser()* sont placés dans un bloc `else` ayant pour pendant une condition de non récursivité : `if (*ppn == NULL)`.

arbre2.c (extrait)

```
void localiser (Noeud** ppn, char* mot)
    /* chercher et insérer un noeud ds le ss-arbre d'adr. (*ppn) */
{
   if (*ppn == NULL) creer (ppn, mot); /* ACTION si "non trouvé" */
   else
   {
      Noeud* pn = *ppn;         /* supprimer une indirection */
      int test = strcmp (mot, pn->texte);
      if (test == 0)  ;         /* ACTION pour le cas "trouvé" */
      if (test < 0) localiser (&pn->fils_g, mot); /* réc.gauche */
      if (test > 0) localiser (&pn->fils_d, mot); /* réc.droite */
   }
}
```

2.2. Classe des variables

Chaque nouvel appel d'une fonction récursive empile un nouvel exemplaire des paramètres formels et des variables automatiques de cette fonction. Chaque niveau d'appel possède ainsi ses propres variables (`pn` *et* `test` *dans l'exemple ci-dessus*). Au retour d'un appel récursif, les variables du niveau précédent ont conservé leur valeur.

Si cette valeur ne doit plus être utilisée, il est inutile d'empiler plusieurs exemplaires de cette variable; tous les niveaux d'appel peuvent utiliser le même exemplaire. Pour garantir la création d'un exemplaire unique de la variable, il suffit de la déclarer ***static*** (ou globale). Ceci est particulièrement à envisager dans le cas d'une variable, structure ou tableau, de taille relativement grande.

> *Exemple.* En ajoutant deux mots `else` au programme précédent, on empêche la réutilisation des variables `pn` et `test` après le retour du premier appel récursif; ces variables peuvent dès lors être déclarées ***static***.

arbre2.c (extrait) – modifié

```
void localiser (Noeud** ppn, char* mot)
    /* chercher et insérer un noeud ds le ss-arbre d'adr. (*ppn) */
{
   if (*ppn == NULL) creer (ppn, mot); /* ACTION si "non trouvé" */
   else
   {
      static Noeud* pn;
      static int test;
      pn = *ppn;              /* supprimer une indirection */
      test = strcmp (mot, pn->texte);
      if (test == 0)  ;       /* ACTION pour le cas "trouvé" */
      else
      if (test < 0) localiser (&pn->fils_g, mot); /* réc.gauche */
      else
      if (test > 0) localiser (&pn->fils_d, mot); /* réc.droite */
   }
}
```

2.3. Elimination de la récursivité

Récursivité terminale dégénérée

(Rappel du chapitre 5.)

Lorsque, comme dans l'exemple ci-dessus, *l'appel récursif est la dernière opération exécutée dans le corps de la fonction* (l'exécution de cet appel n'est suivie que de `return`), on peut écrire une **version itérative** de la fonction, dans laquelle la répétition des appels récursifs est remplacée par un traitement en boucle :

TANT QUE NON (issue non récursive)
EXECUTER simulation de l'appel récursif
FIN-TANT
traitement final d'issue non récursive

La simulation d'un appel récursif consiste ici simplement à *donner de nouvelles valeurs aux paramètres de la fonction*.

La version itérative est plus économe en espace de mémoire (elle supprime l'empilement des paramètres et variables) et en temps d'exécution (elle évite les opérations d'"administration" des appels et retours de fonctions).

Appel récursif simulé

Dans les autres cas, le programmeur doit assurer lui-même les opérations que le compilateur prend à sa charge dans le cas d'un appel récursif : empiler et retirer les paramètres et variables, forcer le branchement de l'exécution sur le début (pseudo-appel) et la poursuite (pseudo-retour) de la procédure récursive.

Une telle programmation utilise des instructions de *branchement* `goto` et une *pile* de données. La présence des instructions `goto` de rupture de séquence empêche l'usage des constructions en boucle (`while`, etc.).

3. Analyse d'une expression arithmétique

Le programme ***calculette*** affiche le résultat d'une expression arithmétique introduite au clavier.

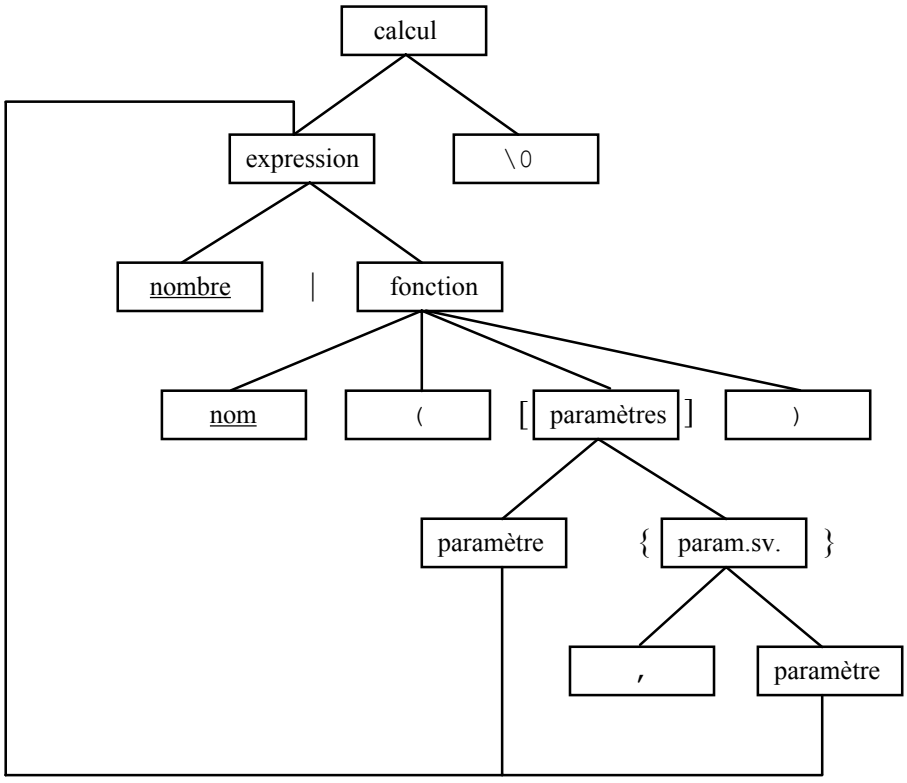
On supposera ici que toute opération est exprimée sous la forme d'un appel de fonction; par exemple, une addition se libellera comme ceci : `plus(2365,4317)`. Cette restriction apporte une double simplification : tous les opérateurs ou fonctions possèdent la même forme syntaxique et la même priorité.

Une expression est soit un nombre, soit un appel de fonction; tout paramètre d'une fonction est lui-même une expression : nombre ou appel de fonction. La définition syntaxique d'une expression apparaît donc récursive.

Une définition syntaxique – une *grammaire formelle* – est habituellement présentée sous la forme d'un ensemble de *règles de production*.

calcul	:=	*expression* '\0'	en séquence
expression	:=	*nombre* | *fonction*	| règles alternatives
fonction	:=	*nom* '(' [*paramètres*] ')'	[ ] membre facultatif
paramètres	:=	*paramètre* {*param.suiv.*}	{ } membre répétitif
param.suiv.	:=	',' *paramètre*	
paramètre	:=	*expression*	

Cet ensemble de règles peut être illustré par l'*arbre de dérivation* ci-dessous.



Le programme est piloté par l'**analyseur syntaxique**, qui vérifie la forme de l'expression.

Ce module demande à la fonction *symb_sv() "symbole suivant"* de l'**analyseur lexical** de lui fournir, un à un, les symboles (nombres, noms, signes spéciaux) formant l'expression. Cette fonction crée un descripteur du symbole lu (catégorie, position, longueur et texte), dont elle renvoie l'adresse à la fonction appelante. Le module d'analyse lexicale comprend aussi la fonction *lire_expr()* qui lit une expression et la normalise en convertissant les majuscules en minuscules.

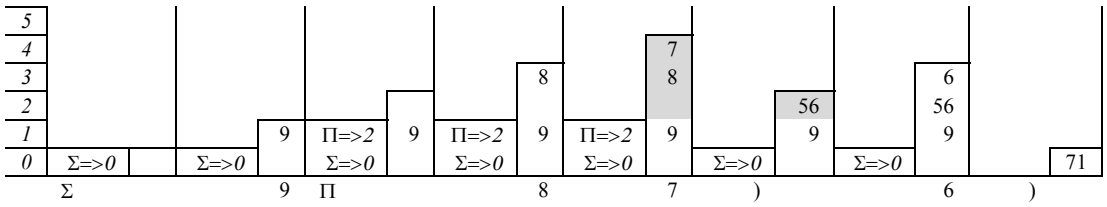
Le module d'**analyse syntaxique** est formé d'une hiérarchie de fonctions *anal_...()*, dont chacune correspond à un noeud de l'arbre de dérivation; certaines concernent un symbole terminal (ex.: *nom* ou *'('*) et d'autres, une règle de composition (ex.: *expression* ou *fonction*). Le résultat d'une analyse peut être un des suivants :

- VRAI : la composition ou le symbole est reconnu valide;
- FAUX : la composition n'est pas reconnue et aucun symbole n'en fait partie – on peut tester l'appartenance du symbole courant à une autre composition;
- FAUX + erreur : certains composants, mais pas tous, ont été reconnus – on conclut que la composition est incorrecte.

C'est encore l'analyseur syntaxique qui active les **routines sémantiques** du programme. Ces routines *exec_...()* exécutent les actions nécessaires au calcul et à l'affichage du résultat. Elles manipulent deux piles : une pile des opérandes ou paramètres et une pile des fonctions appelées :

- lorsqu'un nombre est reconnu, il est placé au sommet de la pile des opérandes;
- lorsqu'un nom de fonction est reconnu, il est placé au sommet de la pile des fonctions, on lui associe la hauteur actuelle de la pile des opérandes, c'est-à-dire la position à laquelle sera placé le résultat de la fonction et au-dessus de laquelle seront empilé ses paramètres;
- lorsqu'est reconnue la fin d'un appel de fonction, on exécute (et enlève) l'opération indiquée au sommet de la pile des fonctions et, au sommet de la pile des opérandes, l'emplacement des paramètres est libéré.

Exemple : `somme(9,produit(8,7),6)`



Ces piles sont réalisées sous la forme de tableaux. Toutes les **fonctions de calcul** programmées possèdent la même interface : elles reçoivent en paramètre l'adresse de la partie du tableau-pile contenant leurs opérandes et placent leur résultat à la position 0 de cette partie de tableau (dans l'exemple ci-dessus, la fonction *produit* reçoit en paramètre l'adresse de la partie grisée de la figure). Une table descriptive des fonctions exécutables indique pour chacune le nombre minimum et maximum d'opérandes admis; la gamme de fonctions exécutables peut donc aisément être étendue.

calc.h

```
#include <stdio.h>
#define msg(texte) puts(texte)      /* affichage des messages */
#define PRIVATE   static      /* pour décl. globales privées */

/*  DECLARATIONS COMMUNES (1) ANALYSE LEXICALE ET SYNTAXIQUE  */

   /* codifications : */
typedef enum {FAUX, VRAI} Logique;
typedef enum {NUL, NBRE, NOM, OPER} Categ_Lexicale;

   /* description d'un symbole : */
typedef struct { char *posit;
                 Categ_Lexicale categ;
                 int longueur;
                 union { float nbre;   /* stocké en format 'float' */
                         char nom[32];
                         char oper;   } texte; }  Symbole;

/*  DECLARATIONS COMMUNES (2) ROUTINES SEMANTIQUES  */

   /* signature d'une fonction de calcul : */
#define routine(fonct)   Logique fonct (int nb_params, float val[])
                         /* nb_params : nombre d'opérandes effectifs
                            val[] : tableau des valeurs paramètres
                                    val[0] reçoit le résultat
                            retourne FAUX en cas d'impossibilité */

   /* description d'une fonction de calcul : */
typedef struct { char nom[32]; routine ((*action)); int minp, maxp; }
               /* (*action) : adr. du corps exécutable de la f()
                  minp/maxp : nbre. de paramètres min/max
                              maxp = -1 <=> max. illimité  */
       Fonction;
```

calc.c

```
/*  CALCULATEUR D'EXPRESSIONS  */

#include "calc.h"

void liste_fonct(void);     /* affiche la liste des f() disponibles */
char * lire_expr(void);     /* renvoie l'adr. du texte ou NULL */
Logique anal_calc(void);    /* renvoie VRAI (1) ou FAUX (0) */

void main(void)   /* lire et calculer une suite d'expr. */
{
   liste_fonct();
   while ( lire_expr() )  anal_calc();
}
```

calc_lex.c

```
/*  ANALYSE LEXICALE  */

#include <stdio.h>            /* sscanf() */
#include <ctype.h>            /* traitement des caractères */

#include "calc.h"
    /*  VARIABLES GLOBALES POUR L'ANALYSE LEXICALE  */
PRIVATE char expr[80+1],     /* texte de l'expression */
            *p_expr;         /* position suivante à examiner */
PRIVATE Symbole symb;        /* descr. du symbole courant */
/*-------------------------------------------------------------------------*/
char * lire_expr (void)    /* lire et préparer une expr */
{
   if (p_expr = (puts("EXPR ou <ctrlZ> :"), gets(expr)))
   {
    /* convertir en minuscules : */
      for (; *p_expr; ++p_expr) *p_expr = tolower(*p_expr);
      p_expr = &expr;   /* se replacer au début */
    /* passer outre des espacements : */
      while (isspace(*p_expr)) ++p_expr;
   }
   return (p_expr);      /* indiquer la 1ère position utilisée */
}
/*-------------------------------------------------------------------------*/
Symbole * symb_sv (void)   /* extraire et décrire le symbole suivant,
                              retourner l'adr. de la descr. du symbole */
    /* N.B. sscanf(..."%n"...) donne le nombre de caractères lus */
{
   while (isspace(*p_expr)) ++p_expr;  /* passer outre des espacements */
   if (*p_expr == '\0')       /* fin de l'expression : */
   {              /* décrire un symbole fictif conventionnel */
      symb.categ = NUL; symb.longueur = 0; symb.texte.oper = '\0';
   }
   else
   if (isdigit(*p_expr))      /* symb. commençant par un chiffre */
   {              /* extraire un nombre : */
      symb.categ = NBRE;
      sscanf(p_expr,"%f%n",&symb.texte.nbre,&symb.longueur);
   }
   else
   if (isalpha(*p_expr))      /* symb. commençant par une lettre */
   {              /* extraire un nom : */
      symb.categ = NOM;       /* N.B. "%[...]" = caract. valides */
      sscanf(p_expr,"%[abcdefghijklmnopqrstuvwxyz_0123456789]%n",
                     &symb.texte.nom,&symb.longueur);
   }
   else                       /* autre caractère */
   {              /* extraire un opérateur : */
      symb.categ = OPER;
      sscanf(p_expr,"%c%n",&symb.texte.oper,&symb.longueur);
   }
   symb.posit = p_expr;      /* position de début du symbole */
   p_expr += symb.longueur;  /* position suivante */
   return (&symb);      /* retourner l'adr. du descr. de symbole */
}
```

calc_stx.c

```
/*  ANALYSE SYNTAXIQUE  */

#include "calc.h"

   /*  VARIABLES ET FONCTIONS GLOBALES POUR L'ANALYSE SYNTAXIQUE  */

Symbole * symb_sv (void);     /* f() de lecture du symbole suivant */
PRIVATE Symbole * pt_symb;    /* ptr vers la descr. du symbole courant */
PRIVATE Logique err_syntaxe;  /* indicateur d'erreur global */

   /*  MACRO-INSTR. DE TRAITEMENT DES ERREURS  */

       /* composition ou symbole non reconnu : */
#define KO              return (FAUX)
       /* composition incorrecte (signale et mémorise l'erreur) : */
#define err(texte)      return (msg(texte), err_syntaxe = VRAI, FAUX)
       /* symbole reconnu (lit le symbole suivant) : */
#define symb_OK         return (pt_symb = symb_sv(), VRAI)
       /* composition reconnue : */
#define OK              return (VRAI)

/*   r := m1  m2      if(!m1) KO; if(!m2) err(); OK;
     r := m1 [m2]     if(!m1) KO; if(!m2 && err_syntaxe) err(); OK;
     r := m1 {m2}     if(!m1) KO; while(m2); OK;
     r := r1 | r2     return (r1 || r2);                              */


   /*  FONCTIONS D'ANALYSE SYNTAXIQUE  */

/* toutes les f() ont la même signature : Logique anal_...(void)
   les f() exec_...(texte_du_symb) sont les routines sémantiques  */

Logique anal_calc(void),      /* f() PRIVATE, sauf anal_calc() */
        anal_debut(void),anal_expr(void),anal_fin(void),
        anal_nbre(void),anal_fonct(void),
        anal_nom(void),anal_parg(void),anal_virg(void),anal_pard(void),
        anal_params(void),anal_param(void),anal_param2(void);

/*  (règle)     calcul := <début> expr '\0'     */
Logique anal_calc (void)
{
   anal_debut();         /* initialiser l'analyse syntaxique */
   if (! anal_expr())                                  /** expr **/
      err(err_syntaxe?"expression invalide":"expression manquante");
   if (! anal_fin()) err("expression invalide");      /** '\0' **/
   OK;
}

/*  (initialisations)  */
PRIVATE Logique anal_debut (void)
{    /**/ void exec_debut(void* pt);
   err_syntaxe = FAUX;   /* annuler l'indicateur d'erreur */
   exec_debut(0);        /* initialiser les routines sémantiques */
   symb_OK;              /* init. OK; lire le 1er symbole */
}
```

```
/*  (règle)     expr := nbre | fonct     */
PRIVATE Logique anal_expr (void)
{
   return ( anal_nbre() || anal_fonct() );      /** nbre | fonct **/
}

/*  (symbole)     '\0'     */
PRIVATE Logique anal_fin (void)
{    /**/ void exec_fin(void* pt);
   if (!(pt_symb->categ == NUL)) KO;
   exec_fin(0);          /* clôturer le traitement sémantique */
   OK;
}

/*  (symbole)     nbre     */
PRIVATE Logique anal_nbre (void)
{    /**/ void exec_nbre(float* pt_nbre);
   if (!(pt_symb->categ == NBRE)) KO;
   exec_nbre(&pt_symb->texte.nbre);     /* traiter le nombre */
   symb_OK;
}

/*  (règle)     fonct := nom '(' [params] ')'     */
PRIVATE Logique anal_fonct (void)
{    /**/ void exec_fonct(void* pt);
   if (! anal_nom()) KO;                                    /** nom **/
   if (! anal_parg()) err("parenthèse gauche manquante");  /** '(' **/
   if (! anal_params() && err_syntaxe)                /** [params] **/
      err("liste de paramètres invalide");
   if (! anal_pard()) err("parenthèse droite manquante");  /** ')' **/
   exec_fonct(0);                       /* traiter la fonction */
   OK;
}

/*  (symbole)     nom     */
PRIVATE Logique anal_nom (void)
{    /**/ void exec_nom(char* pt_nom);
   if (!(pt_symb->categ == NOM)) KO;
   exec_nom(&pt_symb->texte.nom);       /* traiter le nom de f() */
   symb_OK;
}

/*  (symbole)     '('     */
PRIVATE Logique anal_parg (void)
{
   if (!(pt_symb->categ == OPER && pt_symb->texte.oper == '(')) KO;
   symb_OK;
}

/*  (règle)     params := param {param2}     */
PRIVATE Logique anal_params (void)
{
   if (! anal_param()) KO;                              /**  param   **/
   while (anal_param2());                               /** {param2} **/
   OK;
}
```

```
/*  (règle)     param2 := ',' param     */
PRIVATE Logique anal_param2 (void)
{
   if (! anal_virg()) KO;                                  /**  ','  **/
   if (! anal_param()) err("paramètre non reconnu");   /** param **/
   OK;
}

/*  (règle)     param := expr     */
PRIVATE Logique anal_param (void)
{
   return ( anal_expr() );                          /** expr **/
}

/*  (symbole)     ','     */
PRIVATE Logique anal_virg (void)
{
   if (!(pt_symb->categ == OPER && pt_symb->texte.oper == ',')) KO;
   symb_OK;
}

/*  (symbole)     ')'     */
PRIVATE Logique anal_pard (void)
{
   if (!(pt_symb->categ == OPER && pt_symb->texte.oper == ')')) KO;
   symb_OK;
}
```

calc_sem.c

```
/*  ROUTINES SEMANTIQUES  */

#include <string.h>             /* strcmp() */

#include "calc.h"


   /*  TRAITEMENT DES ERREURS  */

PRIVATE Logique err_action;        /* indicateur d'erreur global */
#define err(texte)      (msg(texte), err_action = VRAI)


   /*  VARIABLES GLOBALES POUR LES ROUTINES SEMANTIQUES  */

/* (1) liste des fonctions : */

extern Fonction fonct[];  /* table des descr. de f(), terminée par "" */

int chercher_f (char * pt_nom)   /* validation d'un nom de f() */
   /* renvoie le # indice de la descr. de fonction, ou -1 */
{
   int i;
   for (i=0; fonct[i].nom[0]; ++i)   /* teste la 1ère lettre : "" ? */
      if (strcmp (pt_nom, fonct[i].nom) == 0) return (i);
   return (-1);           /* fonction non reprise dans la liste */
}
```

```
/* (2) piles : */

#define MAX_VAL 32              /* hauteur maximum de la pile */
PRIVATE float val[MAX_VAL];     /* pile des valeurs opérandes */
PRIVATE int v;                  /* niveau suivant */

#define MAX_OPER 16             /* hauteur maximum de la pile */
PRIVATE struct {int i_fonct, i_param;} oper[MAX_OPER];  /* opérateurs */
      /* i_fonct : indice de la descr. de la fonction à exécuter
         i_param : niveau de base dans la pile val[] des opérandes */
PRIVATE int o;                  /* niveau suivant */


   /*  ROUTINES  */
/* toutes les routines ont la même signature :
         void exec_... (ptr vers symbole)      */

/*  (action)     <début> := initialiser un calcul     */
void exec_debut (void * pt)
{    /**/ void exec_nom(char* pt_nom);
   err_action = FAUX;
   v = 0;          /* vider la pile val[] */
   o = 0;          /* vider la pile oper[] */
   exec_nom("AFFICHER");   /* empiler l'appel de la f() "afficher" */
}

/*  (action)     '\0' := afficher le résultat du calcul     */
void exec_fin (void * pt)
{    /**/ void exec_fonct(void* pt);
   exec_fonct(0);       /* exécuter la fonction "AFFICHER" */
}

/*  (action)     nbre := empiler la valeur de l'opérande     */
void exec_nbre (float * pt_nbre)
{
   if (err_action) return;
   if (v < MAX_VAL) val[v++] = * pt_nbre;
   else err("débordement de la pile des opérandes");
}

/*  (action)     nom := empiler l'id. de fonction     */
void exec_nom (char * pt_nom)
{
 /* valider le nom de f() */
   int f = chercher_f(pt_nom);
   if (f < 0) err("fonction inconnue");
   if (err_action) return;
 /* empiler l'indice f de la fonction
           et la hauteur v en réservant l'emplact. du résultat */
   if (o < MAX_OPER && v < MAX_VAL)
     oper[o].i_fonct = f, oper[o++].i_param = v++;
   else err("débordement de la pile des opérateurs");
}
```

```
/*  (action)     ')' := exécuter la fonction en sommet de pile     */
void exec_fonct (void * pt)
{
   if (err_action) return;
   {
      int f = oper[--o].i_fonct;    /* # de la descr. de fonction */
      int p = oper[o].i_param;      /* niveau de base dans val[] */
      int nb_params = v - (p + 1);  /* nbre. de param. effectifs */
      v = p + 1;                    /* nouvelle hauteur de val[] */
      if (nb_params>=fonct[f].minp
         && (fonct[f].maxp<0 || nb_params<=fonct[f].maxp)) /* exécuter */
           err_action = ! (fonct[f].action (nb_params, &val[p]) );
      else err("nombre de paramètres invalide");
   }
}
```

fonct.c

```
/*  TABLE ET EXECUTION DES FONCTIONS DE CALCUL  */

#include <stdio.h>
#include "calc.h"

   /*  MACRO-INSTR. DE TRAITEMENT DES ERREURS  */

        /* opération réussie : */
#define OK              return (VRAI);
        /* opération impossible (signale l'incident) : */
#define err(texte)      return (msg(texte), FAUX);


   /*  ROUTINES DE CALCUL  */

/*  toutes les routines ont la même signature :
           PRIVATE Logique f (int nb_params, float val[])
                  - nb_params : nombre d'opérandes effectifs
                  - val[] : tableau des opérandes
                  val[0] reçoit le résultat
                  retour : VRAI ou FAUX (cf. supra)                */

PRIVATE routine (AFFICHER)    /* affichage du résultat */
{  printf ("\n    %g\n\n", val[1]); OK;  }

PRIVATE routine (somme)
{
   val[0] = val[1];
   while (--nb_params) val[0] += val[nb_params+1];
   OK;
}

PRIVATE routine (produit)
{
   val[0] = val[1];
   while (--nb_params) val[0] *= val[nb_params+1];
   OK;
}
```

```
PRIVATE routine (minimum)
{
   val[0] = val[1];
   while (--nb_params)
      if (val[nb_params+1] < val[0]) val[0] = val[nb_params+1];
   OK;
}

PRIVATE routine (maximum)
{
   val[0] = val[1];
   while (--nb_params)
      if (val[nb_params+1] > val[0]) val[0] = val[nb_params+1];
   OK;
}

PRIVATE routine (soustraction)
{  val[0] = val[1] - val[2]; OK;  }

PRIVATE routine (division)
{
   if (val[2] == 0) err("division par zéro");
   val[0] = val[1] / val[2];
   OK;
}

PRIVATE routine (reste)
{
   if ((long int)val[2] == 0) err("division par zéro");
   val[0] = (long int)val[1] % (long int)val[2];
   OK;
}

PRIVATE routine (absolu)
{  val[0] = (val[1] < 0) ? -val[1] : val[1]; OK;  }

PRIVATE routine (arrondi)
{  val[0] = (long int)(val[1] + 0.5); OK;  }

PRIVATE routine (tronque)
{  val[0] = (long int) val[1]; OK;  }

PRIVATE routine (pourcent)
{
   if (val[2] == 0) err("division par zéro");
   val[0] = val[1] * 100 / val[2];
   OK;
}

PRIVATE routine (tirage)
{
   extern int rand (void);    /* tirage d'un nombre entier aléatoire */
   val[0] = rand(); OK;
}
```

```
/*  TABLE DES DESCRIPTIONS DE FONCTIONS :
          nom, routine, minp, maxp (-1 : nbre. de param. illimité) */

Fonction fonct[] = {
                    {"AFFICHER",AFFICHER,1,1},  /* f() obligatoire */
                    {"plus",somme,2,2},
                    {"moins",soustraction,2,2},
                    {"fois",produit,2,2},
                    {"sur",division,2,2},
                    {"mod",reste,2,2},
                    {"somme",somme,1,-1},
                    {"prod",produit,1,-1},
                    {"min",minimum,1,-1},
                    {"max",maximum,1,-1},
                    {"abs",absolu,1,1},       /* valeur absolue */
                    {"arr",arrondi,1,1},
                    {"tronque",tronque,1,1},
                    {"pct",pourcent,2,2},
                    {"tirage",tirage,0,0},    /* nombre aléatoire */
                    {"",NULL,0,0},   /* convention de fin de table */
                   };

void liste_fonct (void)       /* affiche la liste des f() disponibles */
{
   int i = 1;                        /* passer "AFFICHER" */
   printf ("\t***** FONCTIONS DISPONIBLES ***** \n");
   while (fonct[i].nom[0])           /* teste la 1ère lettre : "" ? */
      printf ("%s  ",fonct[i++].nom);
   printf ("\n\n");
}
```

Commentaire

Les déclarations globales de variables et de fonctions propres à un module, qui ne doivent pas être accessibles de l'extérieur, sont protégées par le qualificatif `PRIVATE (static)`.

La récursivité de l'analyse syntaxique (une expression peut être un appel de fonction, un paramètre de fonction est une expression) est réalisée par l'**appel récursif** de la fonction *anal_expr()*. Il s'agit de **récursivité indirecte**, puisque cet appel n'émane pas de la fonction *anal_expr()* elle-même, mais de la fonction *anal_param()* appelée – indirectement – par *anal_expr()*.

La récursivité de l'analyse sémantique (le résultat calculé par une fonction devient opérande d'une autre fonction) est réalisée par l'emploi d'une **pile de paramètres**.

Exercices

1.

Ecrire une version itérative de la fonction ***localiser()*** du programme *arbre2.c* (page 12-8).

2.

Un texte source en langage C contient habituellement des directives `#include` dont chacune désigne un autre fichier. Dans le texte transmis au compilateur, le texte de chaque fichier inclus remplace la ligne `#include` qui le désigne. Comme un texte inclus peut lui-même contenir des directives `#include`, le procédé est récursif. (*Rappel.* Si, dans une directive `#include`, le nom de fichier est écrit entre les signes `<...>`, ce fichier est à prendre dans un répertoire "standard".)

Rédiger un programme ***developC*** qui, traitant les directives `#include`, écrit dans un fichier le texte à transmettre au compilateur C. La commande d'exécution de ce programme doit lui fournir trois paramètres : le nom du fichier source primaire, le nom du fichier développé, la désignation du répertoire standard. Exemple (MS-DOS) : `developC essai.c essai.lis c:\turboc`

Le module *incl.c* ci-dessous contient les éléments nécessaires à l'identification des fichiers inclus : la fonction *init_design()* enregistre dans une variable globale la désignation du répertoire standard; la fonction *incl()* analyse une ligne de texte – si celle-ci est formée d'une directive `#include`, la fonction renvoie l'adresse de la désignation de fichier à prendre en considération. Ecrire un fichier d'en-tête ***incl.h*** permettant d'utiliser ces fonctions.

incl.c

```
      /*** DEFINITIONS GLOBALES ***/
#define SEPAR   '\\'                /* pour répertoires MS-DOS */
#define PRIVATE     static
PRIVATE char design_fichier[81] = "", * nom_fichier = design_fichier;
    /* la désign. sera construite selon ce schéma :  ...repert\fichier°
                                                      :design_f:nom_f   */

#include <string.h>        /* strcpy() strlen() */
void init_design (char * repert)   /* enregistr. du répert. standard */
{
   if (!repert) return;          /* si ptr NULL : voir déclar. globales */
 /* mémoriser la désignation du répertoire standard : */
   strcpy(design_fichier,repert);
 /* déterminer la position du nom de fichier : */
   nom_fichier = design_fichier + strlen(design_fichier);
 /* si le séparateur \ final manque, l'insérer : */
   if (*(nom_fichier - 1) != SEPAR) *nom_fichier++ = SEPAR;
}

#include <stdio.h>         /* sscanf(), NULL */
char * incl (char * ligne)       /* analyse d'une ligne : #include ? */
{
   char type_include[2];            /* < ou " */
   if (sscanf(ligne," # include %1[<\"]%[^>\"]",type_include,nom_fichier) == 2)
     switch (type_include[0])     /* selon le type d'inclusion */
     {
        case '<': return (design_fichier);  /* adr.-> "...répert\fichier" */
        case '"': return (nom_fichier);     /* adr.-> "fichier" */
     }
   return (NULL);   /* ce n'est pas une ligne #include valide */
}
```

3.

• ***Le paradigme de Programmation Fonctionnelle***

Un langage de ***programmation fonctionnelle*** permet de rédiger un sous-programme, voire un programme complet, sous la forme d'une expression. Or, dans une expression (par exemple : `prix_net+arrondi(prix_net*taux_tva/100)`), les opérations ne sont pas chronologiquement ordonnées et aucune des valeurs produites, qu'il s'agisse de valeurs intermédiaires ou finales, n'est affectée à une variable ...

Un langage de programmation fonctionnelle prédéfinit un certain nombre de fonctions, dont la plupart s'écrivent au moyen d'un **opérateur** (ex.: `+ - * / < > and or`), et des **types** de valeurs de base (ex.: `float`, `int`). Le programmeur ne déclare que des **fonctions** (plus, éventuellement, des constantes et des types de valeurs dérivés des types de base).

La seule opération disponible est l'**application d'une fonction** à un "argument" d'un certain type ou domaine, pour obtenir en résultat son "image" dans un co-domaine ($f : a \rightarrow i$). La méthode de combinaison des opérations est celle de la **composition de fonctions**, consistant à appliquer une fonction (ou un opérateur prédéfini) au résultat d'une autre application de fonction, c'est-à-dire à construire une *expression*.

La programmation fonctionnelle ignore les concepts d'instruction et d'ordonnancement chronologique des instructions (bloc, *if*, *while* ...). Au lieu de la construction *if*, on emploie un opérateur de **fonction conditionnelle**, tel que l'opérateur **`? :`** du langage C; au lieu des constructions en boucle, on utilise des **fonctions récursives**.

La programmation fonctionnelle ignore le concept de *variable*, emplacement modifiable en mémoire; a fortiori ignore-t-elle le concept de tableau. Au lieu de cela, elle manipule des **listes** de valeurs de longueur variable. Ces listes sont toujours parcourues par des fonctions récursives.

Pour permettre le traitement des listes, un langage de programmation doit au minimum définir les **fonctions primitives** suivantes :

– `VIDE()`	rend une liste vide, c'est-à-dire ne comprenant aucun élément
– `AJOUTER(`*élém,liste*`)`	ajoute un élément en *tête* de la liste
	ex.: `AJOUTER('A',AJOUTER('h',VIDE()))` → `"Ah"`
– `PREMIER(`*liste*`)`	rend le *premier* élément de la liste
– `SUITE(`*liste*`)`	rend la partie de la liste qui suit le premier élément

Il doit aussi permettre de présenter une liste sous une forme littérale, qui pourrait être celle-ci :

– `LISTE(`*texte*`)` *ex.:* `LISTE("0.5  1  2.31  1.7  -2")`

Le traitement d'une liste s'effectue toujours au moyen d'une fonction récursive inspirée de ce schéma :

– le traitement programmé s'applique au `PREMIER` élément de la liste;
– l'appel récursif suivant s'applique à la `SUITE` de la liste;
– le bouclage récursif s'arrête lorsque la liste est `VIDE`.

L'intérêt de la programmation fonctionnelle vient de ce qu'elle fait une grande économie de concepts. Elle ne recourt qu'à un très petit nombre de notions mathématiques et ne requiert pas de concepts supplémentaires de technique informatique tels que : tableaux,[83] instructions, procédures, constructions algorithmiques, modes d'adressage direct et indirect, modes de transmission des paramètres, portée des identificateurs, durée de vie des variables, etc. Le prix payé pour cette simplicité est incontestablement celui d'une plus grande abstraction.

- ***Programmation fonctionnelle en langage C***

Tel quel, le langage C permet la programmation fonctionnelle, à l'exception du traitement de listes. (Le programmeur doit se limiter à la seule instruction `return`.)

ex.:
```
unsigned long int factorielle (unsigned long int n)
{ return ( n == 0 ? 1 : n * factorielle (n - 1) ); }
```

Les macro-définitions suivantes permettent de traiter une chaîne de caractères à la façon d'une liste.

```
#define VIDE_T()       ""
#define PREMIER_T(txt) txt[0]
#define SUITE_T(txt)   &txt[1]
```

ex.:
```
int nbre_blancs (char * txt)
{ return strcmp(txt,VIDE_T()) == 0 ? 0 :
         (PREMIER_T(txt) == ' ' ? 1 : 0)
          + nbre_blancs(SUITE_T(txt)); }
```

ex.:
```
int afficher_T (char * txt)
/* comme printf(), indique le nbre de caract. écrits */
{ return strcmp(txt,VIDE_T()) == 0 ? 0 :
         printf("%c",PREMIER_T(txt))
         + afficher_T(SUITE_T(txt));   }
```

- ***Réalisation du support (fonctions primitives)***

On demande de créer, *avec les moyens traditionnels du langage C*, les fonctions primitives qui permettront le traitement fonctionnel de **listes de nombres réels**.

Soit la liste `"0.5  1  2.31  1.7  -2"`. Elle doit, en mémoire centrale, prendre la forme d'une liste chaînée de la façon suivante : → `0.5`→ `1.0`→ `2.31`→ `1.7`→ `-2.0`°

1) Définir les **types** de données nécessaires.

2) Programmer les **fonctions primitives** suivantes :

– `VIDE_N()`	rend une liste vide
– `AJOUTER_N(`*élém,liste*`)`	ajoute un élément en tête de la liste et rend la liste modifiée
– `PREMIER_N(`*liste*`)`	rend la valeur du premier élément de la liste
– `SUITE_N(`*liste*`)`	rend la partie de la liste qui suit le premier élément

(*Remarque*. Une liste peut être représentée par l'adresse de son premier élément.)

[83] Un type structuré est, en programmation fonctionnelle, présenté comme un type *produit cartésien*.

3) Programmer le **constructeur** de liste :

– `LISTE_N(`*texte*`)` *ex.:* `LISTE_N("0.5  1  2.31  1.7  -2")`
`→ 0.5→ 1.0→ 2.31→ 1.7→ -2.0°`

- Dans le texte, les nombres sont séparés par un ou plusieurs espaces.
- Pour extraire du texte le nombre suivant, on utilisera la fonction standard `sscanf()`.
- Pour ajouter ce nombre à la liste en mémoire, employer la fonction `AJOUTER_N()`.
- Cette insertion d'un nouveau nombre doit être suivie d'une correction des chaînages.

• ***Applications***

Rédiger, *en programmation fonctionnelle*, les fonctions suivantes

– **affichage** du contenu d'une liste de nombres,
– calcul de la **longueur** d'une liste de nombres (= nombre d'éléments),
– calcul de la **somme** d'une liste de nombres,
– recherche du **plus petit élément** d'une liste de nombres,
– **jonction** de deux listes, la deuxième se plaçant à la *suite* de la première,
– **inversion** d'une liste (cette fonction utilise la précédente).

4.

Adapter le système de traitement d'un ***arbre AVL*** de manière telle que l'algorithme *localiser()* soit utilisé pour trois opérations différentes : *insérer*, *obtenir* ou *supprimer* un élément.

Parmi ses paramètres, la fonction *localiser()* doit recevoir un code d'opération (**?** *consulter*, **+** *insérer*, **–** *supprimer*).

Un programme interactif doit dérouler de manière répétitive le scénario suivant :

– demander quelle opération doit être exécutée;
– exécuter cette opération;
– à titre de vérification, afficher par la fonction *montrer()* la structure de l'arbre mis à jour.

• ***Suppression d'un élément***

La principale innovation réside dans l'opération de suppression d'un noeud de l'arbre. Cette opération comporte la destruction du noeud (libération de l'espace occupé en mémoire) et la modification de liens (c'est-à-dire de pointeurs) à l'intérieur de la structure générale.

Par rapport à la modification des liens, trois cas sont à distinguer :

– le noeud à supprimer n'a *pas de fils* – dans le noeud père, on doit annuler le pointeur vers le fils supprimé;

– le noeud à supprimer possède *un fils* – on doit rattacher le petit-fils directement à son grand-père ... du bon côté;

– le noeud à supprimer possède *deux fils* – on doit le remplacer par le descendant de rang immédiatement inférieur ou immédiatement supérieur, c'est-à-dire, au choix, par l'élément le plus grand de son sous-arbre gauche ou par l'élément le plus petit de son sous-arbre droit.

Bien entendu, l'équilibre de l'arbre doit être préservé. La question du rééquilibrage après modification doit donc être attentivement étudiée.

Suggestion : il y a peut-être intérêt à décomposer l'opération de suppression en un certain nombre de sous-fonctions : *détruire*, *détacher*, *rattacher* ...

5.

Le programme ***calculette*** peut être adapté pour accepter les opérateurs arithmétiques ordinaires.

- ***Analyse syntaxique***

La grammaire (ensemble des règles de production) doit être modifiée, mais les méthodes de l'analyse syntaxique demeurent inchangées.

Les *parenthèses* autour d'une expression modifient les règles de priorité. L'analyse syntaxique peut traiter cette paire de parenthèses comme un appel de fonction recevant un seul paramètre.

expression	:=	*opérande* {*suite*}	*valeur*	:=	*nombre* | *fonction* | *copie*
suite	:=	*opérateur.2 opérande*	*copie*	:=	'(' *paramètre* ')'
opérateur.2	:=	'+' | '–' | '*' | '/' | '%'	*fonction*	:=	*nom* '(' [*paramètres*] ')'
opérande	:=	*valeur.signée* | *valeur* [84]	*paramètres*	:=	*paramètre* {*param.suiv.*}
valeur.signée	:=	*opérateur.1 valeur*	*param.suiv.*	:=	',' *paramètre*
opérateur.1	:=	'+' | '–'	*paramètre*	:=	*expression*

- ***Routines sémantiques***

La description d'une opération doit indiquer la *position de l'opérateur* ou du nom de fonction par rapport aux opérandes, c'est-à-dire le nombre (0 ou 1) d'opérandes placés avant. Cette information servira à corriger le pointeur vers la pile des opérandes.

La table décrivant les fonctions et opérateurs doit mentionner leur *niveau de priorité*. Le moment d'exécuter une opération de calcul est changé : avant d'enregistrer (empiler) un nouvel opérateur ou nom de fonction, on exécute – à partir du sommet de la pile – toutes les opérations *de l'expression en cours* dont la priorité est supérieure ou égale (en réalité, il se trouve au maximum une opération dans cette situation); de plus, chaque fois que l'analyseur syntaxique atteint la fin d'une expression, il y a lieu d'exécuter les opérations de cette expression qui ne l'ont pas encore été. Pour chaque expression rencontrée, la pile des opérations doit donc contenir un repère de début, que les routines sémantiques peuvent traiter comme une fonction "COPIER", de priorité 0 et sans effet sur la pile des valeurs.

La fonction d'affichage du résultat doit posséder la priorité la plus basse.

---

[84] Plutôt que [*opérateur.1*] *valeur* . L'analyse d'une telle construction, où le premier membre est facultatif, ne saurait comment interpréter (absence ou erreur ?) la non reconnaissance du second membre.

- ***Analyse lexicale***

La fonction *symb_sv()* d'extraction d'un symbole ne tient pas compte du fait que, dans l'expression lue, un nom pourrait être plus long que la zone de réception dans le descripteur. Il y aurait lieu de déterminer d'abord la longueur réelle du nom dans l'expression – `sscanf(..."%*[..]%n"..)` – et de ne copier dans le descripteur que les 31 premiers caractères au maximum.

La même fonction *symb_sv()* peut être adaptée pour accepter une expression s'étendant sur plusieurs lignes. Par exemple, la rencontre du caractère \ pourrait provoquer la lecture d'une nouvelle ligne.

Table des matières

CHAPITRE 1. SURVOL INTRODUCTIF 1-1
1. Définitions de départ *1-1*
2. Etape 1 : élaboration de l'algorithme (analyse) *1-1*
2.1. Etape 1.1 : définir la classe de problèmes 1-1
2.2. Etape 1.2 : définir la méthode de résolution 1-2
2.3. Etape 1.3 : optimiser l'algorithme 1-4
3. Le concept de langage de programmation. Présentation du langage C *1-6*
Histoire du langage C 1-6
4. Etape 2 : réalisation du programme *1-7*
4.1. Etape 2.1 : transformer l'algorithme en fonction programmée 1-7
Règles et habitudes syntaxiques générales 1-8
Formation des mots 1-8
Liste des mots réservés 1-8
4.2. Variante 1-9
4.3. Etape 2.2 : ajouter des opérations d'échange d'information 1-9
Un programme C est un ensemble de fonctions. 1-10
Présentation des fonctions standards de dialogue avec l'opérateur 1-10
Appel de fonction et passation des paramètres 1-11
4.4. Etape 2.3 : organiser les "bibliothèques" de programmes 1-12
Répartition du texte en un ou plusieurs fichiers 1-12
Création du programme exécutable 1-14
5. Un autre exemple *1-15*
5.1. Algorithmes 1-15
5.2. Programme 1-16
5.3. Le concept de module 1-17
6. Supplément. Note sur la documentation des fonctions *1-18*
Exercices *1-19*
CHAPITRE 2. LES VALEURS 2-1
1. Introduction : principes de représentation de l'information *2-1*
1.1. Le système de numération binaire 2-1
1.2. Taille des représentations binaires 2-2
1.3. Interprétation des représentations binaires 2-2
Le concept d'alphabet 2-3
1.4. Exercices 2-4
2. Les types de données scalaires *2-6*
L'opérateur sizeof() 2-6
3. Les constantes littérales *2-7*
3.1 Constantes scalaires 2-7
Liste des caractères non représentables ("escape sequences") 2-8
3.2. Chaînes de caractères 2-8
4. Les variables scalaires (première approche) *2-9*
4.1. Déclaration des variables 2-9
4.2. Usage des identificateurs 2-10
4.3. Initialisation d'une variable 2-10
5. Déclaration des fonctions (première approche) *2-10*
6. Les pointeurs et les constantes adresses (première approche) *2-12*
6.1. Définitions 2-12
6.2. Manipulation des pointeurs et adresses par les fonctions 2-12
7. Les tableaux (première approche) *2-12*
7.1. Mécanismes de base 2-12
Déclaration des tableaux 2-13
Désignation d'un élément de tableau : l'indexation 2-13
Parcours d'un tableau 2-13
Traitement des tableaux par les fonctions 2-14

7.2. Tableaux à plusieurs dimensions ... 2-14
7.3. Chaînes de caractères ... 2-15
7.4. Initialisation des tableaux. Syntaxe de déclaration simplifiée ... 2-17
Exercices ... *2-19*
CHAPITRE 3. LES ENTRÉES ET SORTIES STANDARDS ... 3-1
1. Les flots de données standards ... *3-1*
2. Fonctions d'accès aux flots standards ... *3-1*
Signal de fin de fichier : EOF ... 3-2
2.1. Entrée/Sortie d'un caractère – *getchar()*, *putchar()* ... 3-2
2.2. Entrée/Sortie d'une ligne – *gets()*, *puts()* ... 3-3
2.3. Entrée/Sortie formatée ... 3-3
printf() – "print with format" ... 3-3
scanf() – "scan with format" ... 3-4
Exemples ... 3-6
Problèmes ... 3-6
3. Déviation des flots standards ... *3-7*
3.1. Déviation par le langage de commande ... 3-7
3.2. Déviation programmée – fonction *freopen()* ... 3-7
Exercices ... *3-9*
CHAPITRE 4. LES EXPRESSIONS ... 4-1
1. Définitions ... *4-1*
1.1 Eléments d'une expression ... 4-1
1.2. Catégories d'expressions ... 4-1
1.3. Usage des expressions numériques ... 4-2
Sous-expression en opérande ... 4-2
Instruction ... 4-2
2. Ordre d'évaluation des expressions ... *4-2*
2.1. Emploi des parenthèses ... 4-3
2.2. Priorité et associativité des opérateurs ... 4-3
Priorité ... 4-3
Associativité ... 4-3
Tableau général des opérateurs ... 4-4
3. Le type des opérandes. Conversions de types ... *4-5*
3.1. L'opérateur *sizeof* ... 4-5
3.2. Conversions forcées ... 4-5
Opérations d'affectation ... 4-5
Opérateur de coercition (TYPE) ... 4-6
3.3. Conversions implicites ... 4-6
Promotion entière ... 4-6
Conversion arithmétique ... 4-6
4. Les calculs, au sens large ... *4-7*
4.1. Opérations arithmétiques ... 4-7
4.2. Comparaisons ... 4-9
4.3. Opérations logiques ... 4-10
4.4. Manipulation des bits ... 4-12
5. Les opérations d'affectation ... *4-15*
5.1. Affectation absolue ... 4-15
Erreur fréquente : confusion des opérateurs d'affectation et d'égalité ... 4-16
5.2. Affectation relative ... 4-16
5.3. Incrémentation, décrémentation ... 4-17
6. Les appels de fonctions ... *4-19*
6.1. Passation des paramètres à la fonction appelée ... 4-19
6.2. Renvoi du résultat à la fonction appelante ... 4-20
7. Les expressions algorithmiques ... *4-20*
7.1. Expression conditionnelle (opérateur `?:`) ... 4-20
7.2. Expression séquentielle (opérateur `,`) ... 4-21
8. Effets de bord ... *4-22*

9. Supplément. Quelques fonctions standards ... *4-23*
9.1. Fonctions mathématiques ... 4-23
Liste des fonctions (fichier <math.h>*)* ... 4-23
Gestion des erreurs (fichier <errno.h>*)* ... 4-23
Liste des fonctions (fichier <stdlib.h>*)* ... 4-24
9.2. Gestion des caractères (fichier <*ctype.h*>) ... 4-25
Exercices ... *4-26*
CHAPITRE 5. LE CONTRÔLE DE SÉQUENCE ... 5-1
1. Introduction ... *5-1*
1.1. Concepts ... 5-1
Expressions ... 5-1
Instructions ... 5-1
Constructions ... 5-2
1.2. Conventions générales du langage C ... 5-3
Terminateur d'instruction ; ... 5-3
Bloc { } ... 5-3
Condition ( ) ... 5-4
2. Les constructions itératives ... *5-4*
2.1. while ... 5-4
Simplification de la construction ... 5-6
Programmes hiérarchisés ... 5-6
2.2. for ... 5-8
2.3. do while ... 5-10
3. Les constructions alternatives ... *5-11*
3.1. if ... 5-11
Construction `if` *ou expression conditionnelle ?* ... 5-12
3.2. switch ... 5-13
4. Les instructions de saut ... *5-15*
4.1. break – continue ... 5-15
4.2. return *[expression]* ... 5-16
4.3. goto *étiquette* ... 5-16
5. Fin d'exécution du programme : fonction exit() ... *5-16*
6. La récursivité ... *5-17*
Elimination de la récursivité ... 5-18
7. Programmation par fonctions ... *5-19*
7.1. Méthodes de composition de fonctions ... 5-19
Composition séquentielle ... 5-20
Composition par emboîtement ... 5-20
Composition cachée ("encapsulation") ... 5-20
7.2. Critères de décomposition en fonctions ... 5-20
Conception ascendante ... 5-22
Conception descendante ... 5-22
Exemple ... 5-23
Exercices ... *5-25*
CHAPITRE 6. LE PRÉ-PROCESSEUR ... 6-1
1. Mise en page du programme ... *6-1*
2. Les fichiers d'en-tête .h ... *6-2*
3. Les macro-définitions ... *6-2*
3.1. Macro-définitions constantes ... 6-2
3.2. Macro-définitions paramétrables ... 6-3
3.3. Mécanismes de substitution avancés ... 6-4
Création d'une chaîne de caractères ... 6-4
Concaténation ... 6-5
Redéfinition d'un identificateur ... 6-5
3.4. Compilation conditionnelle ... 6-6
Inclusion conditionnelle ... 6-7
3.5. Remarques sur la gestion des macro-définitions ... 6-7
Options à la compilation ... 6-8

4. Supplément. La démonstration des programmes : assert() *6-8*
Exercices *6-10*
CHAPITRE 7. VARIABLES ET FONCTIONS 7-1
1. Attributs d'une variable *7-1*
1.1. Portée et visibilité 7-1
Portée locale ou globale d'un identificateur 7-1
Usage privé ou public 7-2
Exemples 7-2
Directive d'utilisation 7-4
Règles de visibilité 7-4
Extension de visibilité des déclarations globales (`extern`) 7-4
1.2. Durée de vie et classe d'une variable 7-5
Organisation de la mémoire allouée à un programme 7-6
1.3. Initialisation d'une variable 7-8
Méthode générale 7-8
Initialisation des variables agrégats 7-8
Initialisation des variables automatiques 7-9
1.4. Droit de mise à jour d'une variable 7-10
2. Déclaration des variables et fonctions *7-11*
2.1. Canevas général 7-11
2.2. Déclaration des variables 7-12
Déclaration primaire 7-12
Rappel de déclaration des variables globales 7-13
2.3. Déclaration des fonctions (format ANSI) 7-13
Déclaration primaire 7-13
Déclaration des paramètres formels 7-14
Rappel de déclaration : prototype 7-15
2.4. Déclaration des fonctions (ancienne forme) 7-15
Déclaration primaire 7-15
Rappel de déclaration 7-16
Exercices *7-17*
CHAPITRE 8. LES TYPES CONSTRUITS 8-1
1. Méthodes de construction de types de données *8-1*
1.1. Généralités 8-1
Portée des déclarations de types 8-2
NOTE. Référence à un type construit défini ultérieurement 8-3
1.2. Types codifiés : les énumérations 8-3
Déclaration 8-3
Utilisation 8-4
Portée des déclarations 8-5
Remarque méthodologique 8-5
1.3. Types composites : les structures 8-6
Déclaration 8-6
Opérations (1). Taille d'une structure 8-8
Opérations (2). Désignation des membres 8-9
Opérations (3). Copie de structures 8-10
Initialisation des variables structurées 8-11
1.4. Types alternatifs : les unions 8-11
Déclaration 8-11
Utilisation 8-12
Opérations 8-13
1.5. Les champs de bits 8-13
Déclaration 8-14
Utilisation 8-14
2. Déclaration de noms de types *8-15*
Déclaration et utilisation 8-15
Portée des déclarations 8-16
Exercices *8-17*

CHAPITRE 9. LES POINTEURS 9-1
1. Concepts de base *9-1*
1.1. Définitions 9-1
1.2. Les pointeurs comme paramètres de fonctions 9-2
2. Pointeurs et variables composites *9-3*
2.1. Variables composites et adressage indirect 9-3
2.2. Les variables composites comme paramètres de fonctions 9-4
3. Pointeurs et tableaux *9-5*
3.1. Parenté entre tableaux et pointeurs 9-5
3.2. Calculs d'adresses par déplacement 9-5
3.3. Les tableaux comme paramètres de fonctions 9-7
3.4. Parenté entre chaînes de caractères et pointeurs 9-9
3.5. Note sur les pointeurs comme résultats de fonctions 9-10
4. Les conversions de pointeurs *9-11*
4.1. Pointeurs génériques 9-11
4.2. Pointeurs nuls 9-12
4.3. Exemple 9-12
4.4. Pointeurs sur des objets de types différents 9-13
4.5. Pointeurs et nombres entiers 9-13
5. Synthèse des opérations sur les pointeurs *9-14*
5.1. Opérateurs d'adressage 9-15
5.2. Manipulation des types 9-16
5.3. Opérations arithmétiques 9-16
5.4. Comparaisons 9-17
5.5. Opérations logiques 9-17
5.6. Expression conditionnelle : `expr1 ? expr2 : expr3` 9-18
5.7. Opérations d'affectation 9-18
5.8. Incrémentation, décrémentation 9-19
5.9. Appel de fonction : `f(x,...)` 9-19
6. Quelques fonctions utilitaires *9-20*
6.1. Manipulation des chaînes de caractères (fichier <*string.h*>) 9-20
6.2. Allocation dynamique de mémoire (fichier <*stdlib.h*>) 9-21
6.3. Composition/analyse de texte : *sprintf()*, *sscanf()* 9-23
6.4. Conversion de nombres (fichier <*stdlib.h*>) 9-25
6.5 Interaction avec le langage de commande de l'ordinateur (fichier <*stdlib.h*>) 9-26
6.6 Gestion du calendrier et de l'horloge (fichier <*time.h*>) 9-26
Exercices *9-28*
CHAPITRE 10. LES FICHIERS 10-1
1. Introduction : le concept de fichier *10-1*
Classification des fichiers d'après la nature de leur support 10-1
Classification des fichiers d'après la nature de leur contenu 10-2
2. Connexion et déconnexion d'un fichier *10-2*
2.1. Ouverture d'un fichier – *fopen()* 10-2
2.2. Fermeture d'un fichier – *fclose()* 10-3
2.3. Exemple 10-4
2.4. Les flots standards 10-4
3. Traitement des exceptions *10-4*
3.1. Fonctions d'analyse des exceptions – *ferror()*, *feof()* 10-4
3.2. Retours exceptionnels – *EOF*, *NULL* 10-5
4. Fonctions d'accès à un fichier de texte *10-5*
4.1. Lecture/Ecriture d'un caractère – *fgetc()*, *fputc()* 10-5
4.2. Lecture/Ecriture d'une chaîne de caractères – *fgets()*, *fputs()* 10-6
4.3. Lecture/Ecriture de données formatées – *fscanf()*, *fprintf()* 10-7
5. Fonctions d'accès à un fichier binaire *10-7*
5.1. Lecture/Ecriture d'objets binaires – *fread()*, *fwrite()* 10-7
5.2. Positionnement dans un fichier binaire – *fseek()*, *ftell()* 10-9
5.3. Application : un système de fichiers relatifs 10-11

6. Gestion de la mémoire tampon – fonction fflush() *10-14*
Remarque importante sur les opérations de mise à jour 10-14
Exercices *10-15*
CHAPITRE 11. STRUCTURES DE DONNÉES COMPLEXES 11-1
1. Pointeurs et tableaux *11-1*
1.1. Tableaux à plusieurs dimensions 11-1
Adressage par déplacement paramétrable 11-2
1.2. Tableaux de pointeurs 11-3
Les paramètres de la commande d'exécution 11-4
1.3. Pointeurs de pointeurs 11-5
2. Pointeurs et structures *11-7*
2.1. Structures contenant des pointeurs 11-7
2.2. Listes chaînées 11-8
Liste chaînée selon une structure de file 11-8
Liste chaînée selon une structure de pile 11-11
3. Pointeurs et fonctions *11-12*
3.1. Pointeurs de fonctions 11-12
Syntaxes de déclaration d'un pointeur de fonction 11-14
Syntaxes d'appel d'une fonction 11-14
Fonctions standards de traitement d'un tableau ordonné : qsort(), bsearch() 11-14
3.2. Fonctions à liste de paramètres variable 11-15
Déclaration 11-15
Appel 11-15
Exécution 11-16
Macro-définitions (fichier <stdarg.h>*)* 11-18
3.3. Paramètres passés par référence (simulation) 11-18
Exercices *11-19*
CHAPITRE 12. STRUCTURES RÉCURSIVES 12-1
1. Parcours d'un arbre de recherche binaire *12-1*
Le concept d'arbre 12-1
Représentation d'un arbre binaire 12-1
Opérations 12-2
Arbre AVL 12-5
Commentaire 12-7
2. Règles de programmation des fonctions récursives *12-8*
2.1. Terminaison de l'algorithme 12-8
2.2. Classe des variables 12-8
2.3. Elimination de la récursivité 12-9
Récursivité terminale dégénérée 12-9
Appel récursif simulé 12-9
3. Analyse d'une expression arithmétique *12-10*
Exercices *12-21*